A Beginner's Guide to FreeBasic

Richard D. Clark
Ebben Feagan

A Clark Productions / HMCsoft Book

To all the members of the FreeBasic community, especially the developers.

Acknowledgments

Writing a book is difficult business, especially a book on programming. It is impossible to know how to do everything in a particular language, and everyone learns something from the programming community. I have learned a multitude of things from the FreeBasic community and I want to send my thanks to all of those who have taken the time to post answers and examples to questions.

I would also like to say a big thanks to the developers of FreeBasic. Without your continual hard work in development, we wouldn't have what I consider the premier Open Source BASIC language for today.

The PDF version of this book was created Using Open Office 2.0.2, http://www.openoffice.org/.

About the Book

FBeginner 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 Clark Productions website located at http://rdc.ascii-world.com/ or http://freebasic.hmcsoft.org/.

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. You do not type in the line numbers; they are only for referencing the code in the text. The file name for the program is located after the code segment.

```
1   Option Explicit
2   Cls
3   Print "Hello World From FreeBasic!"
4   Sleep
5   End
```

Listing 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 with a fixed font. 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 using the line numbers referenced in the listing. You do not type the numbers in the editor.

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: program.bas

A potential problem is shown with a Caution paragraph. It is formatted similarly to the Analysis paragraph.

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

Table of Contents

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

1 A Brief Introduction to FreeBASIC

FreeBASIC is a 32-bit BASIC compiler that outputs native code for Microsoft Windows, Linux and DOS via DJGPP. FreeBASIC is also Open Source, which means any one may freely view and edit the source to suit their needs. Quickbasic Compatibility is FreeBASIC's call to fame, for it is the most compatible compiler available.

Differences from QuickBASIC

- **Default (DEF###) type of not explicitly declared variables**
  - FreeBASIC: INTEGER
  - QuickBASIC: Single
- **INTEGER's size**
  - FreeBASIC: 32-bit, use SHORT type for 16-bit integers
  - QuickBASIC: 16-bit
- **Function calling**
  - FreeBASIC: All functions must have been declared, even with CALL.
  - QuickBASIC: With CALL it is possible to invoke prototype-less functions.
- **Arrays not declared**
  - FreeBASIC: All functions must be explicitly declared.
  - QuickBASIC: Arrays are automagically created with up to 10 indexes.
- **Variables with the same names as keywords**
  - FreeBASIC: Not allowed, even with suffixes.
  - QuickBASIC: Allowed if no suffix is used (ie, dim LEFT as integer).
- **Alignment / Padding of TYPE fields**
  - FreeBASIC: Same as in C, use FIELD=constant to change.
  - QuickBASIC: Never done.
- **Fixed-Length strings**
  - FreeBASIC: Real length is the given len plus one (null char), even on TYPE fields. Strings are filled with nulls, so strings can't contain null characters.
  - QuickBASIC: Strings are filled with whitespaces.

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 offscreen pages.
    - All drawing functions can operate on screen as well as on offscreen 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-aloneness: generated EXEs will depend only upon system libs, no external libraries required.
    - As all of FB, also gfxlib is completely multiplatform: 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 frontend)
    - 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, without wrappers**
    - 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 DirectMedia Layer: multimedia library for audio, user input, 3D and 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 FastCGI: 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 s AS 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-bit application, FreeBASIC can compile source code files up to 2 GB long.
    - 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.

2 Numeric 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. These kind of intermittent errors are difficult to find and difficult to 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 numeric 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 3.1 lists all the numeric data types that FreeBasic supports. In the list below, 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
Double	64-bit, floating point, 8 bytes	-2.2E-308 to +1.7E+308
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
Single	32-bit, floating point, 4 bytes	1.1 E-38 to 3.43 E+38
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
Pointer	32-bit, memory address, 4 bytes	Must be initialized at runtime

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.

Pointer Data Types

Pointer data types are unlike the other data types in that they store a memory address and not data. Since pointers are handled differently than regular data types, a separate chapter has been devoted to the subject and will not be examined in this chapter.

Numeric Variables

The numeric data types define what numbers you can work with in your program, but you must create variables to actually hold the numbers. A variable is a name composed of letters, numbers or the underscore character such as MyInteger, or MyInteger2. There are rules for variable naming though, that you must keep in mind.

- 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 conflicts with existing system variable names.
- 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.
- Numbers can be used in a variable name, but cannot be the first character. MyVar1 is legal, but 1MyVar will generate a compiler error.

The compiler creates variables in two ways. The first method, which isn't recommended and has been moved to the QB compatibility mode, is to let the compiler create variables on their first use. When the compiler first encounters a name in a program that is not a reserved word or keyword, it creates a variable using that name with the default data type. In most cases, the default data type is an integer. You can change the default variable type by using the DEF### directive, which will be covered later in this chapter. To enable these features, you must compile your program with either "-lang qb" or "-lang deprecated". The problem with the first-use method is that it is very easy to introduce hard-to-find bugs in your program.

The term “bug” for a software error has a long, and somewhat disputed history. The term predates modern computers and was used to describe industrial or electrical defects. Its use in relation to software errors is credited to Grace Hopper, a pioneer in the field of software design. According to the story, a moth was discovered between two electrical relays in a Mark II computer. Hopper made a note of it in her journal, along with the moth, and the term became associated with software errors after the incident. The moth is now on display at the Smithsonian.[1]

The Dim Statement

The second, and preferred method is to explicitly declare variables in your program using 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 for the variable myInteger. Whenever you manipulate myInteger in a program, the compiler will manipulate the data at the memory location set aside for myInteger. Just as with variable names, there are some rules for the use of Dim as well.

1. **Dim myVar As Integer**. This will create a single integer-type variable.
2. **Dim As Integer myVar, myVar2**. This will create two integer-type variables.
3. **Dim myVar As Double, myVar2 As Integer**. This will create a double-type variable and an integer-type variable.
4. **Dim myVar as Integer = 5**. This will create an integer-type variable and set the value of the variable to 5.
5. **Dim myVar As Double = 5.5, myVar2 As Integer = 5**. This will create a double-type variable and set the value to 5.5, and an integer-type variable and set the value to 5.
6. **Dim Shared as Integer myInt**. This will create a shared (global) variable myInt accessible anywhere within your program.

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.

Shared Variables

As you can see in rule 6 above, using the Dim Shared version of Dim creates a shared variable. This means that any variable you create as Shared can be accessed anywhere within the program’s current module. To put it another way, the scope of a shared variable is module level scope. Scope refers to the visibility of a variable, where you can access a particular variable within a program. There are different levels of scope and each are discussed in detail later in this book.

The number of variables in your program (as well as the program itself) is limited only by your available memory, up to 2 gigabytes.

[1] See WhatIs.com Definitions: http://whatis.techtarget.com/sDefinition/0,290660,sid9_gci211714,00.html

Static Variables

Static variables are used within subroutines and functions and retain their values between calls to the subroutine or functions. The following program demonstrates using a static variable as a counter within a subroutine.

```
1    Sub StaticSub()
2        'Dimension a static variable
3        Static cnt As Integer
4
5        'Increment the count
6        cnt += 1
7        Print "In StaticSub";cnt;" time(s)."
8    End Sub
9
10   'Dimension working variable
11   Dim i As Integer
12
13   'Call sub 10 times
14   For i = 1 To 10
15       StaticSub
16   Next
17
18   Sleep
19   End
20
21
```

Listing 2.1: static.bas

`Analysis:` In line 1 the subroutine StaticSub is defined. A subroutine is code that is executed only when called by the subroutine name which, in this case, is StaticSub. Line 3 dimensions a static variable cnt, which is incremented when the subroutine is called in line 6. Line 7 prints out the current value of cnt to the console screen.

In line 10 a working varible that will be used in the For-Next is declared. Lines 14 through 16 define the For-Next loop which will call StaticSub 10 times. Line 18 waits for a key press and line 19 ends the program.

Running the program in FBIde, you should get the following output.

```
In StaticSub 1 time(s).
In StaticSub 2 time(s).
In StaticSub 3 time(s).
In StaticSub 4 time(s).
In StaticSub 5 time(s).
In StaticSub 6 time(s).
In StaticSub 7 time(s).
In StaticSub 8 time(s).
In StaticSub 9 time(s).
In StaticSub 10 time(s).
```

Output 2.1: Output of static.bas

As you can see from the output, the value of cnt is preserved between calls to the subroutine. 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.

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 will be discussed in detail in the chapter on creating DLLs with FreeBasic.

Data Type Suffixes

You can use QuickBasic style data type suffixes in FreeBasic, although this feature was implemented mainly to support QuickBasic legacy code and is only available when compiling with the “-lang qb” or “-lang deprecated” compiler options. Table 3.2 lists the data type suffixes.

Data Type	Suffix
Byte	b
Short	s
Integer	%
Long	&, l
Ulong	ul
LongInt	ll
UlongInt	ull
Single	!
Double	#
String	$

Table 2.2: Supported Data Type Suffixes

Changing The Default Data Type

As already mentioned, the default data type for an undeclared variable is an integer. The default data type can be changed for a range of variables by using one of the DEF statements. Table 3.3 lists all the DEF statements available in FreeBasic.

Statement	Comment
DEFBYTE a-b	Sets the default data type to byte for undeclared variables starting with letter range.
DEFDBL a-b	Sets the default data type to double for undeclared variables starting with letter range.
DEFINT a-b	Sets the default data type to integer for undeclared variables starting with letter range.
DEFLNG a-b	Sets the default data type to long for undeclared variables starting with letter range.
DEFLNGINT a-b	Sets the default data type to longint for undeclared variables starting with letter range.
DEFSHORT a-b	Sets the default data type to short for undeclared variables starting with letter range.
DEFSNG a-b	Sets the default data type to single for undeclared variables starting with letter range.
DEFSTR a-b	Sets the default data type to string for undeclared variables starting with letter range.
DEFUBYTE a-b	Sets the default data type to ubyte for undeclared variables starting with letter range.
DEFUINT a-b	Sets the default data type to uinteger for undeclared variables starting with letter range.
DEFULNGINT a-b	Sets the default data type to ulongint for undeclared variables starting with letter range.
DEFUSHORT a-b	Sets the default data type to ushort for undeclared variables starting with letter range.

Table 2.3: FreeBasic DEF Statements

The DEF statement will affect all variables that start with a letter in the given range. So if you add DEFDBL m-n to your program, any variable starting with m or n, will default to a double-type. All other variables that start with different numbers will default to an integer-type. A Dim statement will override any DEF statement, so if you declare an integer variable starting with m, it will be an integer, even though the DEFDBL is in effect.

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 2.4: 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 10^5, 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.e-004`.

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 crate 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 integer 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 large integer data you should use a uinteger for positive data or a longint for large negative and positive numbers. 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 right sized data type is critical in large arrays where a few wasted bytes can add up to a large amount of wasted memory.

Option Explicit

You may notice when looking at other people's source code, the compiler directive Option Explicit has been used at the top of each program. A compiler directive is code that instructs the compiler to do something. In the case of Option Explicit, it instructs the compiler to make sure that any variable being used has been properly Dim'ed. Although you can write a program without using Option Explicit and explicitly Dim'ing each variable, you run the risk of introducing nasty, hard-to-find bugs as the following short program illustrates.

```
Dim myInteger as Integer
'set myInteger to a value
myInteger = 5
'Oops we mispelled the variable name
Print "The output of myInteger is";myIntger
'wait for a keypress
Sleep
End
```

Listing 2.2: nooptionexplicit.bas

`Analysis:` In line 1 myInteger is being declared to the compiler. The compiler will use this declaration to allocate space for the variable. In line 3 the variable is being initialized to a value, in this case 3. The Print statement in line 5 will print out the result to the console screen. Sleep, listed in line 7, will wait for a keypress and the End keyword in line 8 will end the program. The End statement isn't mandatory at the end of a program but should be used especially if you want to return an exit code to the operating system.

After typing this program into FBIde and running the program, you should see the following output.

```
The output of myInteger is 0
```

Output 2.2: nooptionexplicit.bas

The output should be 5, but as you can see, the result isn't what was expected or wanted. Since Option Explicit wasn't used in the program, the compiler created a new variable when it encountered the misspelled variable name, myIntger, in the print statement. If this was large and complicated program, you can see how difficult it would be to track this error down and fix it.

Now, add Option Explicit at the top of the program and run it again. You should see the following error message in the results pane of FBIde.

```
Variable not declared, found: 'myIntger'
Print "The output of myInteger is";myIntger
                                    ^
```

Output 2.3: Modified nooptionexplicit.bas

Here the compiler detected the misspelled variable name and informed us of the problem. You should always use Option Explicit in your programs. A few seconds of extra typing will save you hours of frustration.

A Note about Option Explicit

Recent versions of the FreeBASIC compiler do not allow OPTION commands including OPTION EXPLICIT. From now on OPTION EXPLICIT is implied and your programs will behave as if you used OPTION EXPLICIT. It is a good rule of thumb to always implicitly declare your variables, so get in the habit now. To compile a source file that uses OPTION EXPLICIT, either use "-lang deprecated" or remove the OPTION EXPLICIT line from the source file.

A Look Ahead

When working with numeric data, there are times when it becomes necessary to convert one data type to another. FreeBasic offers two conversion methods, implicit and explicit, which you will see in the next chapter.

Excercises

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 hexidecimal number?
7) What Alphabetic letters are allowed in a hexidecimal number?
8) What would the hexidecimal number 1AF be in decimal form?

3 Converting Numeric Data Types

When you work with numbers there will come a time when you will need to convert a variable from one data type to another data type. In FreeBasic there are two types of data conversion. Implicit, where the compiler will automatically convert the data types during an assignment or calculation, and explicit using one of the predefined conversion functions.

Implicit Data Conversion

Implicit data conversion occurs either through an assignment statement, or as the result of a calculation. Where implicit data conversion can cause a problem is in the loss of precision that can result when converting a large data type to a smaller data type. While implicit data conversion eases the programming process somewhat, you want to make sure that the results you are getting are what you expect. You should always check the result of an implicit conversion to make sure the range of values being converted is in the expected range.

The following short program illustrates the conversion that takes place during an assignment statement.

```
1   Dim As Double myDbl
2   Dim As Integer myInt
3   'Set myDbl to a float value
4   myDbl = 5.56
5   'Assign myInt the float, will be converted to int
6   myInt = myDbl
7   Print "myInt ="; myInt
8   Sleep
9   End
10
```

Listing 3.1: assignconv.bas

`Analysis:` In lines 1 and 2 two variable are being declared, myDbl, a double-type varibale and myInt, an integer-type variable. Line 3 is a comment which, as you can see, starts with the ' (single quote) character. In line 4 the double-type variable is being initialized to the floating-point value 5.56. In line 6 the double is being assigned to an integer variable, invoking the implicit conversion. In line 7 the Print displays the result on the console window. In line 8 the program waits for a key press with the Sleep statement and in line 9 the End statement is used to end the program.

Running the program will result in the following output.

```
myInt = 6
```

Output 3.1: assignconv.bas

In the program, the value of myDbl which is a double-type variable, was set to 5.56. When myInt was assigned this float value, the compiler converted the value to an integer, and then rounded it up to 6. Maybe you were just looking for the whole number portion of 5? In this case, your result would be incorrect, although you may not know until later in the program. This type of subtle bug is another one of those problems that are hard to track down and fix.

Caution Even if the compiler is producing the correct result, there is no guarantee that future versions of the compiler will. It may be necessary to change the behavior of the compiler to add features or fix a bug, and you may find yourself with a program that suddenly quits working.

The next little program illustrates the implicit conversion that takes place during a calculation. In this example, the two integer operands are converted to double-types during the division calculation.

```
1   Dim As Double myDbl
2   Dim As Integer myInt1, myInt2
3   'Assign values to myInt1, myInt2
4   myInt1 = 5
5   myInt2 = 3
6   myDbl = myInt1 / myInt2
7   Print "myDbl ="; myDbl
8   Sleep
9   End
10
```

Listing 3.2: calcconv.bas

`Analysis:` Line 1 and 2 in the program are the alternate Dim statement formats. You can use this format to declare multiple variables of the same type. In line 4 and 5, the variables are initialized. In line 6 the / character is the division operator. The result of the division operation will be implicitly converted to a double-type in the assignment statement. Line 7 prints the newly converted value to the screen and in lines 8 and 9, the program will wait for a key press and then end.

Running the program produces the following output.

```
myDbl = 1.666666666666667
```

Output 3.2: calconv.bas

The result is as expected since the double-type has a greater precision than the integer-type. However, consider this program and its output, which demonstrates the implicit rounding during calculations.

```
1   Dim As Double myDbl1, myDbl2
2   Dim As Integer myInt
3   'Assign values to myInt1, myInt2
4   myDbl1 = 5.6
5   myDbl2 = 3.1
6   myInt = myDbl1 / myDbl2
7   Print "myInt ="; myInt
8   Sleep
9   End
10
```

Listing 3.3: calcconv2.bas

`Analysis:` This program is similar to the program in Listing 4.2 except the conversion process is from two double-type variables to an integer. Lines 1 and 2 declare the working variables. Lines 4 and 5 set the values of the two double-type variables. In line 6 the double-type division result is implicitly converted to an integer, resulting in precision loss. Line 7 prints the result to the console window and lines 8 and 9 wait for a key press and end the program.

The output of the program:

```
myInt = 2
```

Output 3.3: calconv2.bas

In this example, the two double-type variables were converted to integers and then the division operation was performed. Since 5.6 was rounded up to 6 during the conversion, the result is 2 rather than 1. This may not be a problem, but you should be aware that these types of conversions occur when working with mixed precision types.

Explicit Data Conversion

There is an alternative to implicit data conversion, explicit data conversion where you use one of FreeBasic’s built-in conversion functions. Since these functions are designed for conversion, they return consistent results and are unlikely to change even if the implicit rules of the compiler change. Even though it is more work, it is always safer to explicitly convert data to the needed data type before carrying out operations on that data.

Numeric Data Conversion Functions

Table 4.1 lists all the conversion functions. Keep in mind that these functions do not check for overflow, so be sure that the value passed to these functions is in the expected range.

Function	Syntax	Purpose	Comment
Cast	B = Cast(datatype, expression)	Convert expression to data-type listed.	Datatype is any of the available FreeBasic data-types. Expression is a numeric value, variable or expression.
CByte	B = CByte(expression)	Convert expression to byte.	Expression must be in the range –128 to 127.
CDbl	B = CDbl(expression)	Convert expression to double.	Expression must be in the range of -2.2E-308 to +1.7E+308.
Cint	B = Cint(expression)	Convert expression to integer.	Expression must be in the range of –2,147,483,648 to 2,147,483,647.
CLng	B = CLng(expression)	Convert expression to long. Long is an alias for integer.	Expression must be in the range of –2,147,483,648 to 2,147,483,647.
ClngInt	B = ClngInt(expression)	Convert expression to long int.	Expression must be in the range of –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
CShort	B = CShort(expression)	Convert expression to short.	Expression must be in the range of -32768 to 32767.
CUByte	B = CUByte(expression)	Convert expression to unsigned byte.	Expression must be in the range of 0 to 255.
CUInt	B = CUInt(expression)	Convert expression to unsigned integer.	Expression must be in the range of 0 to 4,294,967,296.
CULngInt	B = CULngInt(expression)	Convert expression to long unsigned integer.	Expression must be in the range of 0 to 18,446,744,073,709,551,615.
CUShort	B = CUShort(expression)	Convert expression to unsigned short.	Expression must be in the range of 0 to 65535.
CSng	B = SCng(expression)	Convert expression to single.	Expression must be in the range of 1.1 E-38 to 3.43 E+38.

Table 3.1: FreeBasic Conversion Functions

When examining conversion functions it is important to understand how the function rounds with both negative and positive numbers. The FreeBasic conversion functions round numbers up, making positive number more positive and negative numbers more negative. The following program illustrates the rounding with Cast and Cint, as well as demonstrating that the conversion functions will work with expressions as well as numbers.

```
1   Dim as Double myDouble1, myDouble2, myDouble3
2   Dim as Integer myInt
3
4   'Set variable ranges
5   myDouble1 = 143.5
6   myDouble3 = 143.4
7   myDouble2 = -143.5
8   myInt = 12
9
10  'Show cast in action
12  Print "** Cast **"
13  Print "Double1 ";myDouble1;" cast to integer ";Cast(Integer, myDouble1)
14  Print "Double2 ";myDouble2;" cast to integer ";Cast(Integer, myDouble2)
15  Print "Double3 ";myDouble3;" cast to integer ";Cast(Integer, myDouble3)
16  Print "Expression ";myDouble1;" + ";myInt;" cast to double ";
17  Print Cast(Double, myDouble1 + myInt)
18  Print
19
20  'Show cint in action
21  Print "** CInt **"
22  Print "Double1 ";myDouble1;" cint to integer ";CInt(myDouble1)
23  Print "Double2 ";myDouble2;" cast to integer ";CInt(myDouble2)
24  Print "Double3 ";myDouble3;" cast to integer ";CInt(myDouble3)
25  Print "Expression ";myDouble1;" + ";myInt;" cast to integer ";CInt(myDouble1 + myInt)
26  Print "Expression ";myDouble2;" + ";myInt;" cast to integer ";CInt(myDouble2 + myInt)
27  Print
28
29  Sleep
30  End
31
32
```

Listing 3.4: convert.bas

`Analysis:` This program demonstrates the built-in FreeBasic conversion functions. Lines 1 and 2 declare the working variables using the alternate format of the Dim. Two tables of information are printed to the console in line 12 to 18 and lines 21 to 27. In lines 13 to 16, the result of the Cast function is printed directly to the screen, as is the Cint function in lines 22 to 26. Normally the values would be saved in another variable to be used in the program. Line 16 ends with a semi-colon. This instructs the compiler not to print a carriage return after the print statement, so that when the value of Cast is printed in line 17, the value will appear on the same line as the data printed in line 16. The program is closed in the usual way using Sleep and End.

Running the program will produce the output shown below.

```
** Cast **
Double1  143.5 cast to integer  144
Double2 -143.5 cast to integer -144
Double3  143.4 cast to integer  143
Expression  143.5 +  12 cast to double  155.5

** CInt **
Double1  143.5 cint to integer  144
Double2 -143.5 cast to integer -144
Double3  143.4 cast to integer  143
Expression  143.5 +  12 cast to integer  156
Expression -143.5 +  12 cast to integer -132
```

Output 3.4: convert.bas

As you can see from the program, you can use both numbers and expressions in the conversion functions. Keep in mind that if you pass an expression to a conversion function, the evaluated expression must not exceed the limit of the target data type, or an overflow will occur. Looking at the output you can see that the rounding for middle values is up or greater in magnitude. 143.5 rounds up, making the resulting number more positive and –143.5, more negative.

Using Conversion Functions in Macros

One area where the conversion functions listed in Table 4.1 are useful is in the creation of macros. A macro is a small piece of executable code that the compiler inserts into the program at designated areas. Macros are defined using the #Define compiler directive. #Define has the format `#Define` *name symbol*. When the compiler encounters *name* within the program code, it is replaced with *symbol*.

The following program creates a macro that packs a screen row and column number into the high and low bytes of a uinteger. The conversion function Cint is used since you will not always know the data type being passed to the macro, and you want to be sure that you are working with known quantities.

```
'Macro created by v1ctor
#define MAKDWORD(x,y) (cint(x) shl 16 or cint(y))

Dim myInt As Uinteger
Dim As Integer i, cnt

'Store row 5 column 5 in a single uinteger
myInt = MAKDWORD(5, 5)

'Set the width and height of the console window
Width 80, 25
'Print column headers
cnt = 1
For i = 1 To 80
    'Print columns as 12345678901...
    If cnt = 10 Then
        cnt = 0
    End If
    Locate 1, i
    'Convert to string so we don't get leading space
    Print Str(cnt)
    'Increment our counter
    cnt += 1
Next
'Print row headers
cnt = 2
For i = 2 To 25
    'Row numbers will be like col numbers
    If cnt = 10 Then
        cnt = 0
    End If
    Locate i, 1
    'Convert to string so we don't get leading space
    'We need the semi-colon so a line feed isn't printed
    'on line 25 which would scroll screen.
    Print Str(cnt);
    'Increment our counter
    cnt += 1
Next

'Print out string on saved location
Locate Hiword(myInt), Loword(myInt)
Print "We stored the screen location in a single uinteger!"

```

```
45  Sleep
46  End
47
48
```

Listing 3.5: makdword.bas

`Analysis:` In line 2 the #define compiler directive is used to create a macro. A macro is an snippet of executable code that the compiler will insert into the code when it finds the macro name, MAKDWORD. In lines 4 and 5 the working variables are declared. In line 8, the variable myInt is initialized with the macro previously defined. The macro stores the screen locations, row 5 and column 5 in the high and low word of the integer variable.

In line 11 the Width statement is used to set the console screen to 80 columns and 25 rows. The variable cnt, which will be used to print out the column and row headers, is set to 1. Lines 14 through 24 prints out the column headers using a For-Next loop. Line 19 uses the Locate statement to set the cursor position and line 36 prints out the value. The numbers will range from 1 to 0 and then repeat.

In line 36 the Str function is used to convert the numeric value of cnt to a string so that the Print statement will not insert a leading space. Print inserts a leading space when printing numeric variables so they can accommodate a leading negative sign. Lines 26 through 39 print out the row headers in the same fashion as the column headers.

In line 42 the two functions Hiword and Loword are used to retrieve the high and low bytes of the integer variable, which contain the row and column respectively. Line 45 and 46 wait for a key press and end the program in the usual fashion.

The macro is defined in line 2 takes two parameters, x and y. The compiler will substitute `MAKDWORD(x,y)` with `(cint(x) shl 16 or cint(y))` replacing `x` with the first number and `y` with the second number. Cint is used within the macro so that no matter what values are passed through x and y, the macro will always be working with integers. The operators **or** and **shl** will be discussed later in the book.

The Locate statement expects a row number and column number to position text on the screen. In line 42 we are using the two functions Hiword and Loword to return the high (most significant) and low (least significant) words of the integer myInt, which contain the row and column numbers respectively.

Running the program will produce the following output.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
2
3
4
5   We stored the screen location in a single uinteger!
6
7
8
```

```
9
0
1
2
3
4
5
6
7
8
9
0
```

Output 3.5: makdword.bas

#Define is fully explored in the chapter on symbolic constants later in the book.

Decimal Rounding Functions

FreeBasic has two built-in rounding functions for use with decimal numbers. Table 4.2 lists the functions along with their explanations.

Function	Syntax	Comment
Int	B = Int(expression)	Returns an integer value less than or equal to the input.
Fix	B = Fix(expression)	Returns the integer part of the input.

Table 3.2: FreeBasic Rounding Functions

As with the other conversion functions, this should be tested with both negative and positive numbers to see how the function behaves. The program intfix.bas listed below illustrates the results returned by the functions as well as their use with calculations.

```
'Create some double variables
Dim As Double myDouble1 = -5.5, myDouble2 = 5.9
'Show rounding on negative and positive values
Print "Doubles:", myDouble1, myDouble2
Print "Int:", Int(myDouble1), Int(myDouble2)
Print "Fix:", Fix(myDouble1), Fix(myDouble2)
Print
'Try some calculation expressions
myDouble1 = -15.78
myDouble2 = 22.12
Print "Expression:",myDouble1;" +";myDouble2;" = "; Str(myDouble1 + myDouble2)
```

```
12  Print "Int:", Int(myDouble1 + myDouble2)
13  Print "Fix:", Fix(myDouble1 + myDouble2)
14
15  'Wait for keypress
16  Sleep
17  End
18
```

Listing 3.6: intfix.bas

`Analysis:` Line 2 uses the alternative syntax for Dim to declare and initialize the working variables. Line 4 through 6 display the results of the conversion function suing both positive and negative numbers. Line 9 and 10 set the values for the variables to be used in calculations. Lines 12 through 14 display the results of the conversion functions using calculations so FreeBasic resolves the calculations before invoking the functions. Lines 17 and 18 end the program in the usual way.

The result of running the program is shown in Output 4.6.

```
Doubles:      -5.5          5.9
Int:          -6            5
Fix:          -5            5

Expression:   -15.78 + 22.12 = 6.34
Int:           6
Fix:           6
```

Output 3.6: infix.bas

Notice that in the output Int returns a –6 for the input –5.5, since 6 is less than –5.5, and 5 for 5.9 since 5 is less than 6. Fix on the other hand truncates the decimal and returns the integer portion of the double. As you can see both functions will also work with an expression, in this myDouble1 + myDouble2. Which function you would use depends on what behavior you would need with negative numbers, as they both return similar results with positive numbers.

A Look Ahead

In addition to the built-in conversion functions, there are a number of functions available in the C Runtime Library, which is the subject of the next chapter.

Exercises

1) How do you stop your program from printing a carriage return at the end of a line?
2) What function would you use to convert an unknown value to a Double?
3) What function would you use to convert an unknown value to an Integer?
4) What would be the result of the equation 5 / 2 if assigned to an Integer?
5) How do the functions Int() and Fix() differ in their results?

4 Introduction to the C Runtime Library

One of the original motivations for creating FreeBasic was to provide an easy-to-use wrapper for the C Runtime Library according to Andre Victor, the language creator. FreeBasic is as advanced as it is today because many of the functions are wrappers for the runtime functions, which simplified the creation of the language.

Since FreeBasic uses the C runtime library, you can use it as well in your programs, and there are a number of very useful conversion functions available. The function definitions are contained in the header file crt.bi, and you gain access to those functions by using the #Include directive.

Caution Not all the functions listed in the various bi files in the crt folder have been implemented in FreeBasic. If you get an error similar to the following [**path/filename].o:fake:(.text+0x16c): undefined reference to `rint'**, where path/filename is your program's path and filename, then the function you are trying to use is not supported. This is a linker error and indicates that the linker was not able to find the referenced function. There are also a number of functions and structures that will only work in Windows and some that are GNU Extensions and will only work on Linux.

The #Include Directive

The syntax for the #Include directive is **#Include [Once] "filename.ext"**. Remember that a compiler directive instructs the compiler to do something, and in his case, it instructs the compiler to include the file "filename.ext" when compiling. When the compiler reads this directive, it stops reading the current file and starts reading the included file. If the included file also has an #Include directive, then that file is read in as well. You can have up to 16 nested includes per program. To use the runtime conversion functions in your own program you would add **#Include Once "crt.bi"** at the top of your program.

The file extension *bi* stands for "basic include" and is usually declaration statements and/or subroutines and functions. The crt.bi file is located in the *inc* folder of your FreeBasic installation. If a bi file is not located in the folder where the program is being compiled, the compiler will look in the inc folder for the file. If it can't find the file, it will issue an error. You can also include a path with the filename if you keep your include files in a common folder besides the inc folder.

If you look at the contents of crt.bi you see that the file simply includes other files and references the library that contains the actual function code.

```
#ifndef __CRT_BI__
#define __CRT_BI__

#ifdef __FB_WIN32__
# inclib "msvcrt"
#endif

#include once "crt/string.bi"
```

```
9    #include once "crt/math.bi"
10   #include once "crt/time.bi"
11   #include once "crt/wchar.bi"
12   #include once "crt/stdlib.bi"
13   #include once "crt/stdio.bi"
14   #include once "crt/io.bi"
15   #include once "crt/errno.bi"
16
17   #endif ' __CRT_BI__
```

Listing 4.1: crt.bi

`Analysis:` Crt.bi contains a number of compiler directives, the statements that start with the # character to control the compilation process. In line the directive #ifndef means “if not defined”. If the symbol __CRT_BI__ has not already been defined, then line 2 will define the symbol. The #ifdef directive in line 4 means “if defined”. If the compiler is running on a Win32 machine, the symbol __FB_WIN32__ will be defined which will include the library "msvcrt" on line 5 with the #inclib directive which means “include library”. The #endif on line 6 closes the opening #ifdef on line 4. Under Windows, FreeBasic uses the Microsoft Runtime which is contained in the dynamic link library (DLL) msvcrt.dll.

Line 8 through 15 include each of the individual declaration files that make up the C runtime library. The “#include once” directive ensures that the individual include files are only added once, even if another module includes them. The #endif on line 17 is used to close the #ifndef on line 1.

Notice that the #Include directive use the Once modifier which tells the compiler that this file should only be loaded once, even if a subsequent file uses the same #Include. This is to prevent duplicate definition errors. It is always good practice to use the Once modifier in your own includes.

Notice that the different files being included are located in the inc/crt folder. This is an example of setting the path for the #Include if they should be in a different folder. Many of the third-party library bi files are in their own subfolders, just to keep things a bit more organized. The conversion functions we will be examining in this chapter are contained the file crt/math.bi.

The #Inclib directive above the include list tells the compiler what library contains the functions. Library files are compiled subroutines, functions and symbols that define the functionality of the library. Lib files located in the *lib* folder of your FreeBasic installation. They are usually named filename.a for static libraries and filename.dll.a for dynamic link libraries. If you look in the lib folder you will also see some filename.o files. These are object files. Object files contain pre-parsed code that can be linked to an executable file or to a library file.

Caution If you open one of the bi files in the inc folder, which you may need to do from time to time to see what parameters a function requires or to see the return type of a function, be sure not to change anything in the file. You should always open the files in read-only mode if your editor provides that functionality.

The Declaration Statement

If you look at math.bi you will see a number of Declaration statements. A declaration statement is a function prototype that tells the compiler that there is a function definition somewhere in the program code, or in a library or an external dll. This *forward reference* is used so that the compiler won't complain if the function or sub is called before it finds the actual code for the function. Once you understand how to interpret a declaration statement, you will be able to use any function in the runtime library.

To understand the components of a declaration statement, here is the declaration statement for the ceil function in math.bi. The ceil function rounds a decimal value up to the nearest integer. That is, 1.5 would become 2.

```
Declare function ceil cdecl Alias "ceil" (byval as double) as double
```

You can break this declare down into the following functional sections.

- **Declare**: This is the keyword for a declare statement.
- **Function**: This can either be Function, to indicate that this procedure returns a value, or Sub, which indicates that the procedure does not return a value.
- **Ceil**: This is the name of the function and the name that you would use in your code. Do not use the name after Alias.
- **Cdecl**: This is the function's method of passing arguments. When calling an external function, parameters must be pushed on the stack in the correct order so that the function can pop the parameters off the stack correctly. Cdecl stands for C Declare, and passes the arguments from right to left using the C convention of parameter passing. The **Pascal** calling convention pushes arguments from left to right and is used when calling functions created in the Pascal language. The **Stdcall** calling convention pushes arguments from right to left and is used in Basic libraries and Windows API.
- **Alias**: This is an alternate name that is used by the linker when your code is linked to other languages. You use an alias in case there is a function in the target language that has the same name as your function.
- **( )**: The values inside the parenthesis are the parameters that the function or sub expects. Ceil expects one parameter, a double-type value. If the function expects multiple parameters, then commas will separate them. A parameter can be any of the FreeBasic data types or a composite type. Many of the runtime functions require pointers to other data types or data structures.
- **ByVal**: The byval (by value) keyword indicates that the value being passed to the function is a copy of the data, and not the actual data itself. This is to prevent accidentally changing the actual data within a function. You can also use **ByRef** (by reference) which passes the address of the data to the function, and any changes in the parameter are reflected in the actual data. ByVal is the default behavior in current versions of the compiler.

- **As [data type]**: The As keyword after the parameter list indicates the return type of the function. Subs do not have a return type so there isn't an As clause after the parameter list.

Notice that the ceil function returns a double and not an integer, even though the conversion function rounds off the decimal portion. This is to prevent integer overflow for very large numbers. Even though ceil, and the other runtime conversion functions return a double you can implicitly convert them to integers by simply assigning the return to an integer variable or using the Cast function. Be sure to use an integer data type with a large enough range for the result.

To use Ceil in your program you would write `myDouble = Ceil(aDouble)` or `myInt = Ceil(aDouble)`. All declaration statements follow the same pattern as Ceil.

DLLs created with Visual Basic 6tm and below are COM objects and not standard DLLs. In order to use VB6 DLLs with FreeBasic you will need to create a COM interface to access the DLL functions. Creating a COM interface is a complicated topic and is beyond the scope of this book.

Runtime Conversion Functions

Table 5.1 lists some of the more useful conversion routines contained within the runtime library along with he syntax and some comments about the function.

Function	Syntax	Comment
Ceil	B = Ceil(double-type expression)	Ceil returns the nearest integer greater than expression.
Floor	B = Floor(double-type expression)	Floor returns the nearest integer less than expression.
Modf	B = Modf(double-type expression, double-type pointer)	Modf returns the fractions part of expression and the integer part of expression in the double-type pointer. The integer part of expression is rounded towards zero.

Table 4.1: Runtime Conversion Functions

It would require another book to examine all the functions in the runtime library. The runtime functions presented in this book were selected because of their usefulness and to supplement FreeBasic's built-in functions.

Modf deserves special mention. A function can only return a single value, the value defined in the As clause after the parameter list. This does not mean however that you cannot return more than one value from a function. Any additional values a function returns must be returned through the parameter list. One way to do this is to use the Byref keyword on parameters. Any changes made to a byref parameter changes the actual data. However, you cannot use byref on external libraries, due to differences in calling procedures, so pointers are used instead. There are two ways to pass pointer data to a function; by explicitly declaring a pointer data type which you will see in the chapter on pointers, or by using the AddressOf operator which is explained here.

The AddressOf Operator @

When the compiler creates a variable, it allocates a portion of memory equal to the size of the data type. For an integer it would allocate 4 bytes and for a double it would allocate 8 bytes. Each memory location has an address associated with it so that the CPU can move data into and out of the memory location. When the compiler allocates the memory, the operating system returns an address for the memory location so that the compiler can access that location. When you are working with variables you are actually working with an addressable memory location in the computer. You don't have to worry about the address since the compiler handles all those details for you; you can just use the variable in your program.

However, when you need to call a function like Modf that requires a pointer to a variable, then you need to know the address of the variable so you can pass it to the function. Why can't you just use the variable name, since it is an alias for the memory location? The answer is in the declaration of Modf shown in listing 5.3.

```
declare function modf cdecl alias "modf" (byval as double, byval as double ptr) as double
```

Listing 4.2: Modf Declaration

Notice that the second parameter is defined as byval and the data type is a pointer (ptr) to a double-type variable. Remember that byval means “make a copy of the contents of the parameter and use it in the function.” If you were to simply use the variable name, you would be passing the contents of the memory location, whatever is in the variable,and not its address.

When you use a regular variable in your program, you are working with the actual data in the memory location associated with the variable. That is, a variable = data. If you define myInt as an integer and set it equal to 5, when you print myInt to the screen you will see 5. The pointer data type is unique in that what it contains isn't data, that is a value like 134, rather it contains the address of a memory location. A pointer = memory address. If you were to print a pointer to the screen you would see a number that doesn't make much sense, since what you are looking at isn't the data in the memory location, but the address of the data in memory.

When you first create a pointer, it doesn't point to anything. This is called a NULL pointer. There are a number of ways to initialize the pointer, but one method is to use the AddressOf operator. If you create a pointer variable such as myIntPtr you could initialize that pointer with code like `myIntPtr = @myInt`. The @ is the AddressOf operator, and this code snippet sets myIntPtr to the address of myInt. In other words, an initialized pointer variable and the AddressOf operator both return the same value, a memory address.

We can use this concept in the Modf function. Rather than creating a pointer variable, we can create a double-type variable and then use this variable with the AddressOf operator for the return parameter. Remember that you were able to use expressions with the built-in numeric conversion functions. Using the AddressOf operator with a variable is an expression which will be evaluated and the result, the variable address, will be passed along to the function.

The following short program illustrates this concept.

```
1   'Need to include the runtime library
2   #include "crt.bi"
3
4   'Set up some variables
5   Dim As Double myDouble, FracPart, IntPart
6
7   'Set the value we want to convert
8   myDouble = 12.56
9   'Call the function. Notice that we are using the addressof
10  'operator @ on IntPart.
11  FracPart = modf(myDouble, @IntPart)
12  'Get the result.
13  Print "myDouble = ";myDouble
14  Print "Fractional part:";FracPart
15  Print "Integer part:";IntPart
16
17  Sleep
18  End
19
20
```

Listing 4.3: addressof.bas

Analysis: In line 1 the crt.bi is included in this program so that all the supported functions will be available to the program. In line 5 all of the working variables are declared. In line 8, the variable to be operated on is set to a decimal value. The Modf function is called in line 11 using the AddressOf operator to retrieve the integer portion of the return value. The fractional portion of the return value is loaded into the FracPart variable. Line 13 through 15 print out the values returned from the function. The program is closed in the usual way in lines 17 and 18.

In line 5 we dimension the variables we will use: myDouble is the value we want to convert, FracPart is the value returned by the function and IntPart is the return value that will be passed through the second parameter, the pointer. Notice in line 11 we are using the AddressOf operator in conjunction with our regular double-type variable.

Here is the output of the program.

```
myDouble =  12.56
Fractional part: 0.560000000000001
Integer part: 12
```

Output 4.1: addressof.bas

The return value of the function, .56, is the fractional part and the integer part 12, is the value returned through the parameter. The series of zeros ending in a 1 after the

.56 is the result of the precision conversion you will get with a double value. Remember in the previous sections it was mentioned that double and single data types are prone to rounding errors. Here you see why this is true.

Caution When you are dealing with a pointer parameter, you must be sure to pass the correct data type to the parameter. In the case of Modf, it is expecting a double-type pointer which is 8 bytes. If you pass the incorrect pointer type to a function, it may not do anything at all, or it may try to write to 8 bytes of memory. If the function tries to write to more memory than has been allocated you will see the infamous General Protection Fault message and your program will be shut down. Always use the proper pointer type with pointer parameters. General Protection Fault is a Windows term and is equivalent to a Segmentation Fault on Linux.

Testing the Runtime Conversion Functions

As you did with the other conversion functions, you should test these functions with both positive and negative numbers to see how they behave.

```
1   'Need to include the runtime library
2   #include "crt.bi"
3
4   'Set up some variables
5   Dim As Double myDouble = 12.56, myDouble2 = -12.56
6
7   'Show the various conversion functions with
8   'positive and negative numbers.
9   Print "Ceil with";myDouble;" returns ";Ceil(myDouble)
10  Print "Ceil with ";myDouble2;" returns ";Ceil(myDouble2)
11  Print
12  Print "Floor with";myDouble;" returns ";Floor(myDouble)
13  Print "Floor with ";myDouble2;" returns ";Floor(myDouble2)
14  Print
15
16  'Wait for a keypress
17  Sleep
18  End
19
20
```

Listing 4.4: crtfunc.bas

`Analysis:` Since these are C runtime functions, the crt.bi is included in line 2. Line 5 shows the declaration of the working variables in the program, once again using the alternate Dim syntax. Lines 9 through 14 print out the return values of the functions with both positive and negative numbers. Lines 17 and 18 close the program.

Line 2 in the program includes the crt.bi which is required for using any of the runtime functions. In line 5 the two working double-type variables are dimensioned and initialized. As you can see, calling the runtime functions is exactly the same as calling any of the FreeBasic built-in functions. Running the program will produce the following output.

```
Ceil with 12.56 returns  13
Ceil with -12.56 returns -12

Floor with 12.56 returns  12
Floor with -12.56 returns -13
```

Output 4.2: crtfunc.bas

As you can see in the output, Ceil returns the nearest integer greater than the value. 13 is greater than 12 and -12 is less negative than -12.56. Floor does just the opposite; it returns the nearest integer less than the value. 12 is less than 12.56 and -13 is more negative than -12.56.

A Look Ahead

Normally when you use numbers in your program, you are going to do some sort of calculation with them. In the next chapter you will see FreeBasic's arithmetic operators, learn about operator precedence and explore the bits that make up a number.

Exercises

1) What is the extension of files that normally contain function declarations for libraries?

2) What is the difference between the Ceil() and Floor() functions in the C runtime library?

3) What compiler directive is used to add additional files before compilation?

4) What is a common cause of a Segmentation Fault or General Protection Fault?

5 Arithmetic Operators

Arithmetic Operators

The Table 6.1 lists all the arithmetic operators in FreeBasic. In the table below, the operators will work on two or more expressions, where expression is a number, variable or another expression. The square brackets in the syntax column indicate optional additional expressions.

Function	Syntax	Comment
+ (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.
* (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 evaluates to zero (0), then a runtime 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 runtime error will occur.
^ (Exponentiation)	B = expression^value	Returns the result of raising expression to the power of value. That is 2^2 = 2*2.
MOD (Modulo)	B = expression MOD	Returns the remainder of

Function	Syntax	Comment
	expression	the implicit division of the expressions as an integer result. If expression is a decimal value, expression is rounded before the division.
- (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.
( ) (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.

Table 5.1: Arithmetic Operators

You should be familiar with most of these operators from math class. Integer division is used when you are not concerned about the remainder of the division process. The Mod operator has several uses including executing some code only at certain times, and for creating wrap-around functions such as are likely to occur when a sprite reaches the edge of the screen and needs to wrap around to the other side.

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. Table 6.2 lists the precedence rules for the arithmetic operators. The order of precedence is the same order as the listing; that is the top row has the highest precedence, while lower rows have lower precedence.

Operator
( ) (Parenthesis)
^ (Exponentiation)
- (Negation)
*, / (Multiplication and division)
\ (Integer division)
MOD (Modulus)
SHL, SHR (Shift bit left and shift bit right)
+, - (Addition and subtraction)

Table 5.2: Precedence of Arithmetic Operators

The SHL and SHR operators will be discussed in the next section, Bitwise Operators. They are included here to show where they fall in the precedence rules.

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 and 33 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 (is used 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 forced evaluation.

```
Option Explicit

Dim As Integer myInt

'Let compiler evaluate expression according to precedence
myInt = 5 + 6 * 3
Print "Expression 5 + 6 * 3 = ";myInt

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

'Wait for keypress
Sleep
End
```

Listing 5.1: precedence.bas

`Analysis:` As always we open the program with Option Explicit in line 1 and declare the working variables in line 3. In line 6 the compiler evaluates the math expression according to precedence rules and stores the result in myInt, which is printed to the console window in line 7. In line 10, parenthesis are used to force the evaluation of the expression which is printed to the console screen in line 11. Lines 14 and 15 close the program in the 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 5.1: precedence.bas

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

```
Option Explicit

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


```

```
26   Sleep
27   End
```

Listing 5.2: sameprecedence.bas

`Analysis:` In line 3 the working variable myInt is declared, since the program uses the Option Explicit directive in line 1. In line 6 the expression has the + operator first and the – second, while in line 11 the order has been reversed to test the evaluation order. The results are printed in lines 8 and 13. Line 16 and 21 use the * and / operators on the same manner to test the order of evaluation and the results are printed in lines 18 and 23.

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 5.2: Output of sameprecedence.bas

As you can see from the output each expression has been evaluated from left to right. In expression 1, 3 + 5 is 8 and subtracting 4 equals 4. In expression 2, 3 – 5 is equal to -2, and adding 4 equals 2. In expression 3, 6 * 2 is 12 and divide that by 3 and you get 4. In expression 4, 6 /2 is 3 and multiplies by 3 results in 9. The program shows that operators that have the same precedence level are 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

Operator	Syntax	Comment
		assign to B.
\=	B \= expression	Integer divide B by expression and assign to B.

Table 5.3: Shortcut Arithmetic Operators

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`.

Binary Number System

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 )
```

Listing 5.3: Snippet from Windows Message Loop

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 store 65535 ids in a single word, using an unsigned data type, or 32767 ids for a signed data type. 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 data type is negative or positive using the Two's Complement form of notation. To represent a negative number, the positive value of the number is negated, that is all the 1's are changed to 0 and the 0's are changed 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 1111 1011. 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.

```
1    Option Explicit
2
3    Dim As Byte myByte
4    Dim As String myBits
5
6    myByte = 5
7    'Get the binary form of number
8    myBits = Bin(myByte)
9    'Append some leading zeros so print line up
10   myBits = String(8 - Len(myBits), "0") & myBits
11   'Print out nybbles with a space between so is is easier to read
12   Print "myByte =";myByte;" which is binary ";
13   Print Mid(myBits, 1, 4) & " " & Mid(myBits, 5, 4)
14
15   myByte = -5
16   'Get the binary form of number
17   myBits = Bin(myByte)
18   'Append some leading zeros so print line up
19   myBits = String(8 - Len(myBits), "0") & myBits
20   'Print out nybbles with a space between so is is easier to read
21   Print "myByte =";myByte;" which is binary ";
22   Print Mid(myBits, 1, 4) & " " & Mid(myBits, 5, 4)
23
24   Sleep
25   End
```

Listing 5.4: signbit.bas

`Analysis:` Lines 3 and 4 declare the working variables, a byte that will represent the actual number and a string that will represent the binary value. In line 6 the myByte is set to 5. In line 8 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. In line 12 the numeric value of myByte is 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.

In line 13 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 in line 13 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.

Line 15 sets myByte to -5 and lines 17 through 22 format the output and display 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 5.3: 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.

A Look Ahead

There are times when you need to manipulate individual bits and bytes of a variable. FreeBasic has a rich set of bitwise operators and macros that you will see in the next chapter.

6 Bitwise Operators

FreeBasic includes a number of operators that manipulate the bits of a number. The following table lists the bitwise 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 (Bitwise negation)	B = NOT expression	NOT 0 = 1 NOT 1 = 0	Inverts operand bit; turns a 1 into 0 and a 0 into 1.
And (Bitwise 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 (Bitwise 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 is 1.
Xor (Bitwise exclusion)	B = expression XOR expression	0 XOR 0 = 0 1 XOR 0 = 1 0 XOR 1 = 1 1 XOR 1 = 0	Result bit is 1 if operand bits are different.
Eqv (Bitwise 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 (Bitwise 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: Bitwise 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.

```
Option Explicit

Dim As Byte myByte = 5

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

```
7    'Apply NOT operator
8    myByte = Not myByte
9    'Value after NOT operation
10   Print "NOT myByte: ";myByte," Binary: ";Bin(myByte)
11   'Add 1 after NOT operation
12   myByte = myByte + 1
13   'Print result = -5 in decimal and binary
14   Print "myByte + 1: ";myByte," Binary: ";Bin(myByte)
15
16   Sleep
17   End
```

Listing 6.1: not.bas

`Analysis:` In line 3 the working variable myByte is declared and initialized using the alternate format of the Dim statement. Line 6 prints out the value of myByte in both decimal and binary. In line 8 the NOT operator is used to negate all the bits in the integer. The result of the operation is printed to the console window in line 10. In line 12, 1 is added to the result as required by the two's complement method. The result of this operation id printed in line 14. The program is closed in the usual 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 6.1: 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.

```
1    Option Explicit
2
3    'Declare working variable and mask value
4    'The mask will test the 3rd bit position, i.e. 4
5    Dim As Byte myByte = 5, Mask = 4
6
```

```
7    'Print decimal and binary values
8    Print "Testing 3rd bit position (from right)"
9    Print "myByte:";myByte," Binary: ";Bin(myByte)
10   Print "Mask:  ";Mask," Binary: ";Bin(Mask)
11
12   'Check to see if 3rd bit is set
13   If (myByte And Mask) = 4 Then
14       Print "3rd bit is 1"
15   Else
16       Print "3rd bit is 0"
17   End If
18   Print
19
20   'The mask will test the 2nd bit position, i.e. 2
21   Mask = 2
22   'Print decimal and binary values
23   Print "Testing 2nd bit position (from right)"
24   Print "myByte:";myByte," Binary: ";Bin(myByte)
25   Print "Mask:  ";Mask," Binary:  ";Bin(Mask)
26
27   'Check to see if 2nd bit is set
28   If (myByte And Mask) = 4 Then
29       Print "2nd bit is 1"
30   Else
31       Print "2nd bit is 0"
32   End If
33
34   Sleep
35   End
```

Listing 6.2: and.bas

`Analysis:` In line 5 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. Lines 8 through 10 print out the heading, decimal and binary values for the variables. The If statement in line 13 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.

Line 21 sets the mask value to 2 to test for a 1 in the second bit position, which is 2^1 or 2. Line 23 through 25 print out the header, decimal and binary values of the variables. :ine 28 uses the AND operator to test for a 1, and since binary 5 has a 0 in this position, the program will execute the code immediately following hr Else keyword.

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 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.

```
Option Explicit

'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
```

```
17  If (myByte And Mask) = 2 Then
18      Print "myByte contains 2"
19  Else
20      Print "myByte doesn't contains 2"
21  End If
22  Print
23  'Set the mask value to 4
24  Mask = 4
25  'Print decimal and binary values
26  Print "Testing for 4"
27  If (myByte And Mask) = 4 Then
28      Print "myByte contains 4"
29  Else
30      Print "myByte doesn't contain 4"
31  End If
32  Print
33  'Set the mask value to 8
34  Mask = 8
35  'Print decimal and binary values
36  Print "Testing for 8"
37  If (myByte And Mask) = 8 Then
38      Print "myByte contains 8"
39  Else
40      Print "myByte doesn't contain 8"
41  End If
42
43  Sleep
44  End
```

Listing 6.4: or.bas

`Analysis:` Line 4 declares the working variables. In line 7, myByte is set to 2 and in line 8, that value is combined with 4 using the OR operator. Lines 10 and 11 print out the decimal and binary values of myByte. In line 14, the mask is set to 2 and in lines 18 through 22 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. Lines 24 through 31 use the same procedure to test for the value of 4. Since myByte contains a 4, the program will print out the text "myByte contains a 4". In lines 34 through 41, myByte is tested for 8, which is does not contains so the code in line 40 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
```

Output 6.2: 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.

```
Option Explicit

Dim As Integer myInt1 = 5, myInt2 = 10

Print "myInt1 = ";myInt1;" myInt2 = ";myInt2
Print "Swapping values..."
```

```
7    myInt1 = myInt1 Xor myInt2
8    myInt2 = myInt1 Xor myInt2
9    myInt1 = myInt1 Xor myInt2
10   Print "myInt1 = ";myInt1;" myInt2 = ";myInt2
11
12   Sleep
13   End
```

Listing 6.5: xor.bas

Analysis: In line 3 the working variables are declared and initialized. Line 5 prints out the initial variable values. Lines 7 through 9 carry out the XOR operation. In line 7, MyInt1 is XORed with myInt2 to get an intermediate value which is stored in myInt1. In line 8, the XOR operation returns the original value of myInt1 which is stored in myInt2. The third XOR operation in line 9 then returns the value of myInt2 from the intermediate value that was stored in myInt1 in line 7. Line 10 prints 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 6.3: 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 bitwise operator, but it can be used to see if two expressions are equivalent as the following program demonstrates.

```
Option Explicit

#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.
```

```
12  Print "Statement (myInt1 < 5) eqv (myInt2 > 1) ";
13  If (myInt1 < 5) Eqv (myInt2 > 1) = True Then
14      Print "is equivalent."
15  Else
16      Print "is not equivalent."
17  End If
18  Print
19
20  'Both statements are false so are equivalent.
21  Print "Statement (myInt1 > 5) eqv (myInt2 < 1) ";
22  If (myInt1 > 5) Eqv (myInt2 < 1) = True Then
23      Print "is equivalent."
24  Else
25      Print "is not equivalent."
26  End If
27  Print
28
29  'One is true, the other false so statement
30  'is not equivalent.
31  Print "Statement (myInt1 > 5) eqv (myInt2 < 1) ";
32  If (myInt1 > 3) Eqv (myInt2 < 1) = True Then
33      Print "is equivalent."
34  Else
35      Print "is not equivalent."
26  End If
37
38  Sleep
39  End
```

Listing 6.6: eqv.bas

`Analysis:` In lines 3 and 4, the values 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.

In line 6 the working variables are declared and initialized. Lines 12 through 18 test the first expression. The If statement in line 13 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. In line 22, both of the expressions are False. 0 EQV 0 is True, so this statement is also True and will print out the affirmative. In line 32, 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) eqv (myInt2 > 1) is equivalent.

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

Statement (myInt1 > 5) eqv (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 bitwise 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 bitwise 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 bitwise operators operate on either the True or False values of complete expressions to avoid evaluation problems.

Shortcut Bitwise Operators

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

Operator	Syntax	Comment
Imp=	B Imp= C	This is the same as B = B Imp C.

Table 6.2: Shortcut Bitwise 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 0100 which 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.

```
Option Explicit

Dim As Uinteger myInt = 1, i

'Multiply by powers of 2
Print "Shifting left..."
```

```
7    Print "myInt = ";myInt
8    For i = 1 To 8
9        myInt = myInt Shl 1
10       Print "myInt = ";myInt
11   Next
12   Print
13   'Divide by powers of 2
14   Print "Shifting right..."
15   Print "myInt = ";myInt
16   For i = 1 To 8
17       myInt = myInt Shr 1
18       Print "myInt = ";myInt
19   Next
20
21   Sleep
22   End
```

Listing 6.8: shlr.bas

`Analysis:` Line declares the working variables, myInt which will be he value that is shifted, and i for use in the For-Next loop. Line 8 through 11 shift myInt left 16 times, 1 bit position each time, and prints the result. Line 15 through 19 then shift the variable right 16 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 6.4: 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.

Bitwise 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 (1 shl (bit_number))) > 0)
Bitset	B = Bitset(variable, bit_number)	#define Bitset(x, bit_number) ((x) or (1 shl (bit_number)))
Bitreset	B = Bitreset(variable, bit_number)	#define Bitreset(x, bit_number) ((x) and not (1 shl (bit_number)))

Table 6.4: Bitwise Macros

The Hiword macro returns the leftmost two bytes of an integer and the Loword macro returns the the rightmost two bytes. The Hibyte macro returns the leftmost eight bits of a 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 Bitwise operators will only work correctly on byte and integer-type data. A single or double-type variable that is passed to a bitwise 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.

A Look Ahead

In addition to the arithmetic and bitwise operators, FreeBasic has a set of mathematical functions which is the subject of the next chapter.

7 Mathematical Functions

FreeBasic has a number of mathematical functions which are listed in the following table.

Function	Syntax	Comments
ABS (Absolute Value)	B = Abs(expression)	Returns the unsigned value of expression. This is an overloaded function so expression can resolve to an integer, longint or double.
ACOS (ArcCosine)	B = Acos(expression)	Returns the ArcCosine of a double-type expression in radians. Expression must be in the range of -1 to 1.
ASIN (ArcSine)	B = Asin(expression)	Returns the ArcSine of a double-type expression in radians. Expression must be in the range of -1 to 1.
ATAN2 (ArcTangent of Ratio)	B = Atan2(expressionV, expressionH)	Returns the ArcTangent of of ratio of double-type expression1 / expression2 in radians. ExpressionV is the vertical component and expressionH is the horizontal component, where both expressions are double-type values.
ATAN (ArcTangent)	B = Atan(expression)	Returns the ArcTangent of a double-type expression in radians. Expression must be in the range of -Pi/2 to Pi/2.
COS (Cosine)	B = Cos(expression)	Returns the Cosine of a double-type expression in radians. Expression must be in the range of -1 to 1 and is the angle measured in radians.
EXP (E Exponent)	B = Exp(expression)	Returns e (approx 2.716) raised to the power of a double-type expression.
LOG (Logarithm)	B = Log(expression)	Returns the natural Logarithm (base e) of a double-type expression.
SGN (Sign)	B = Sgn(expression)	Returns the sign of a double-type expression. If expression is greater than 0 then Sgn returns a 1. If expression is 0, then Sgn returns 0. If expression is less than 0, then Sgn return -1.
SIN (Sine)	B = Sin(expression)	Returns the Sine of a double-type expression in radians. Expression must be in the range -1 to 1.
SQR (Square Root)	B = Sqr(expression)	Returns the square root of a double-type

Function	Syntax	Comments
		expression. Expression must be greater than or equal to 0.
TAN (Tangent)	B = Tan(expression)	Returns the Tangent of a double-type expression in radians. Expression is the angle measured in radians.

Table 7.1: Mathematical Functions

Trigonometric Functions

All of the trigonometric functions return the measured angle in radians not degrees. To convert degrees to radians you would use the formula radian = degree * PI / 180. To convert radians to degrees you would use the formula degree = radian * 180 / PI. PI is the ratio of the circumference of a circle to its diameter and can be calculated using the Atn function with the formula PI = 4 * Atn(1.0). The following short program converts an angle in degrees to an angle in radians.

```
1    Option Explicit
2
3    'Calc the value of Pi
4    Const Pi = 4 * Atn(1)
5
6    Dim As Double deg, rad
7
8    'Get the angle in degrees
9    Input "Enter an angle in degrees";deg
10
11   'Calculate the radian
12   rad = deg * Pi / 180
13
14   'Print out the values
15   Print
16   Print "Pi:";Pi
17   Print "Degrees:";deg
18   Print "Radians:";rad
19
20   Sleep
21   End
```

Listing 7.1: degrad.bas

`Analysis:` In line 4 a symbolic constant is declared for PI using the Atn function. Line 6 declares the working variables. In line 9, the user inputs the degrees to convert to radians. Line 12 calculates the radian angle based on the conversion formula listed above. Lines 15 through 18 print out the various values. The program is then ended in the usual way.

Running the program will produce the following output.

```
Enter an angle in degrees? 90


Pi: 3.14159265358979
Degrees: 90
Radians: 1.5707963267949
```

Output 7.1: Output of degrad.bas

There are a number of on-line tutorials that you can access to brush up on your trig. A quick Google search on the term trigonometry brought up hundreds of sites that offer basic to advanced tutorials on the various trig functions that FreeBasic supports. Randy Keeling has written two basic tutorials on trigonometry in the Community Tutorials section of the FreeBasic wiki. At the time of this writing, the wiki was located at http://www.freebasic.net/wiki/wikka.php?wakka=FBWiki.

Absolute Value and the Sgn Function

You would use the Abs function when you are interested in the absolute magnitude of a number. The absolute value of a number is the distance that number is from zero on the real number line. 3 is three units from 0 so its absolute magnitude is 3. -3 is also three units from 0 so its absolute magnitude is also 3.

The Sgn function, called the signum function in math, is related to the Abs function since you can express the absolute value of a number by using the Sgn function. Sgn returns -1 for a negative number, 0 for 0 and 1 for a positive number. To calculate the absolute value of a number you would use the formula |a| = a * Sgn(a), where the vertical lines indicate absolute value. If we replace a with -3 in the formula, you get |-3| = -3 * Sgn(-3), or |-3| = -3 * -1. Since multiplying a negative with a negative returns a positive, the result is |-3| = 3. Replacing a with 3, the formula would evaluate to |3| = 3 * 1, or |3| = 3.

One use of the Sgn function in programming is determining the relative positions of two points on the screen. If you have two points, A(5, 6,) and B(3, 4), where the first number is the row and the second number is the column, you can determine the relative position by moving the origin to point A and looking at the sign of point B. The origin (0, 0) of the screen is located in the upper left corner. To move the origin to point A you need to subtract 5 from the row and 6 from the column of each point giving A(0, 0) and B(-2, -2). You can now use the Sgn function on the row and column components of B.

- If Sgn(B.row) = -1, then the row is above A.
- If Sgn(B.row) = 0, then the row is in the same row as A.
- If Sgn(B.row) = 1 then the row is below A.
- If Sgn(B.column) = -1 then the column is to the left of A.
- If Sgn(B.column) = 0 then the column is in the same column as A.
- If Sgn(B.column) = -1 then the column is to the right of A.

Since B is located at row -2 and column -2 after adjusting the origin, the Sgn function will return -1 for the row and -1 for the column. This puts point B above and to the left of of point A.

Logarithms

Logarithmshave a wide range of uses in mathematics, physics and biology. For example, the decay rate of radioactive particles is logarithmic in nature. Logarithms are a way of expressing exponentiation. For the following formula $A^b = x$ can be expressed as Log(x) = b where the base of the log is a. The two most commonly used logarithms are the natural log, which uses e (2.7182818284590452354...) as the base, and log base 10 or the common log. The Log function in FreeBasic uses the natural log, but any log base can be expressed by the formula logarithm = Log(number) / Log(base)[2].

One property of logarithms is that they can be used to multiply or divide two numbers together. This was a common use of logarithms before the invention of calculators. The following program uses the logarithm method to multiply two numbers together.

```
Option Explicit

Const base10 = Log(10)

Dim As Double l1, l2, lt, al

'Calculate the log of 2.5 for base 10
l1 = Log(2.5) / base10
'Calculate the log of 5.6 for base 10
l2 = Log(5.6) / base10

'Add the logarithms
lt = l1 + l2

'Get the antilog, which is our base (10)
'raised to the sum of the log values
al = 10^lt

'Print results
Print "2.5 * 5.6 = ";2.5 * 5.6
Print "Log method = ";al

Sleep
End
```

Listing 7.2: log.bas

[2]For an explanation of the natural log and e see http://www.physics.uoguelph.ca/tutorials/LOG/logbase.html.

Analysis: Line 3 defines a symbolic constant, base10, that will be the base of the logarithm calculations. Using a constant in this manner reduces the number of calculations the program must perform. Instead of using two Log(10) function calls, only one is used in the constant definition. Line 5 declares the working variables, l1 an l2 which are the two Logarithms of the numbers 2.5 and 5.6 and the sum of the logs, lt. al will be the antilog value.

Lines 8 and 10 calculate the base 10 log of the two numbers. In line 13 the logs are added together. Line 17 calculates the antilog, which is the base, in this case 10, raised to the power of the sum of the logs. Line 20 prints the result using the multiplication operator, and line 21 prints the result using the log method.

The program is closed in the usual way in lines 23 and 24.

Running the program produces the following result.

```
2.5 * 5.6 =  14
Log method =  14
```

Output 7.2: Output of log.bas

As you can see from the output, adding the logarithms of the two numbers and then calculating the antilog of the sum produces the same result as multiplying the two numbers.

The C Runtime Library Math Constants and Functions

The C Runtime Library (CRT) has a number of math constants and functions that you can use in your programs. To use these functions you would add the line `#Include Once "crt.bi"` to your program. The following table lists the math constants available in math.bi.

CRT Math Constants

Constants	Value	Comment
M_E	2.7182818284590452354	The base of natural logarithms.
M_LOG2E	1.4426950408889634074	The logarithm to base 2 of M_E.
M_LOG10E	0.43429448190325182765	The logarithm to base 10 of M_E.
M_LN2	0.69314718055994530942	The natural logarithm of 2.
M_LN10	2.30258509299404568402	The natural logarithm of 10.
M_PI	3.14159265358979323846	Pi, which is the ratio of a

Constants	Value	Comment
		circle's circumference to its diameter.
M_PI_2	1.57079632679489661923	Pi divided by two.
M_PI_4	0.78539816339744830962	Pi divided by four.
M_1_PI	0.31830988618379067154	The reciprocal of pi, that is 1/pi.
M_2_PI	0.63661977236758134308	Two times the reciprocal of pi.
M_2_SQRTPI	1.12837916709551257390	Two times the reciprocal of the square root of pi.
M_SQRT2	1.41421356237309504880	The square root of two.
M_SQRT1_2	0.70710678118654752440	The reciprocal of the square root of two which is also the square root of 1/2.

Table 7.2: CRT Math Constants

In the example programs the value of PI was calculated. However, as you can see, the value of PI is available in the CRT, so it is much more efficient to use the defined value M_PI in your programs, rather than making the compiler perform a calculation for the value. The same reasoning applies to the base 10 logarithm calculation that was used in log.bas. Instead of calculating the base 10 log you can use the constant value M_LN10.

The various versions of the PI constants are commonly used values in calculations as are the versions of the square root of 2. The square root of 2, like PI, is an irrational number and is the length of the hypotenuse of a right triangle with sides that have a unit length of 1. The square root of 2 is related to PI with the formula sqrt(2)=2sin(PI/4). The following program displays both the calculated value of the square root of 2 and the CRT value for comparison.

```
Option Explicit

#Include Once "crt.bi"

Dim As Double sqr2

sqr2 = 2 * Sin(M_PI/4)

Print "Calculated Square root of 2:";sqr2
Print "CRT value:"; M_SQRT2

Sleep
```

```
13   End
```

Listing 7.3: sr2.bas

`Analysis:` Line 3 includes the CRT declaration file, crt.bi, which will enable the program to use the math constants and functions. The `#Include Once` directive ensures that the file is only included once in the program, even if other modules also include the file. Line 5 declares the working variable which will hold the result of the calculation in Line 7. Line 9 and 10 displays the calculated value as well as the CRT defined value for comparison. The program is closed in the usual way.

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

```
Calculated Square root of 2: 1.41421356237309
CRT value: 1.4142135623731
```

Output 7.3: Output of sr2.bas

As you can see from the output, the calculated value has a slightly higher precision than the constant value, but the difference is too small to really matter in most cases. To save the compiler a calculation, and thereby speeding up your program, use the defined value in the CRT.

Selected CRT Math Functions

There are a number of math functions declared in the CRT. Table 8.3 list three of the functions.

Function	Syntax	Comment
Hypot	B = Hypot(x as double, y as double)	Hypot returns the hypotenuse of sides x and y.
Log10	B = Log10(x as double)	Log10 returns the base 10 log of x.
Pow	B = Pow(base as double, power as double)	Pow returns base raised to power.

Table 7.3: Selected CRT Math Functions

The log program in Listing 8.2 can be modified to use the CRT functions.

```
Option Explicit

#Include Once "crt.bi"

Dim As Double l1, l2, lt, al
```

```
6
7    'Calculate the log of 2.5 for base 10
8    l1 = Log10(2.5)
9    'Calculate the log of 5.6 for base 10
10   l2 = Log10(5.6)
11
12   'Add the logarithms
13   lt = l1 + l2
14
15   'Get the antilog, which is our base (10)
16   'raised to the sum of the log values
17   al = Pow(10, lt)
18
19   'Print results
20   Print "2.5 * 5.6 = ";2.5 * 5.6
21   Print "Log method = ";al
22
23   Sleep
24   End
```

Listing 7.4: log2.bas

Analysis: Line 3 includes the crt.bi declaration file so that the runtime functions will be available to the program. Line 5 declares the working variables as in the previous program. In lines 8 and 10, the previous calculation has been replaced with the CRT function log10. Line 13 adds the logarithms, which is the same as multiplying the two numbers, and saves the result in the variable lt. In line 17 the Pow function is used rather than the exponentiation operator to calculate the antilog. Lines 20 and 21 print out the results for comparison. The program is closed in the usual way.

When you run the program you will see that the output is the same as in the previous version.

```
2.5 * 5.6 =  14
Log method =  14
```

Output 7.4: Output of log2.bas

Using the CRT functions can make your program a bit more simple, and a bit less error prone. Since FreeBasic uses the Microsoft runtime library the functions are stable and reliable, and many include optimizations that make the functions perform better than you could achieve by hand coding.

A Look Ahead

In addition to the standard numeric variables, FreeBasic also has the pointer data type, which is the subject of the next chapter.

8 Pointer Data Type

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 ImageCreate, 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 an 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.

```
Option Explicit

'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))
```

```
8    'Print the address
9    Print "Pointer address:";myPointer
10   'Add some meaningful data
11   *myPointer = 10
12   'Print the contents-will not be garbage
13   Print "Memory location contains:";*myPointer
14   'Deallocate the pointer
15   Deallocate myPointer
16
17   Sleep
18   End
```

Listing 8.1: basicpointer.bas

`Analysis:` In line 4 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. Line 7 uses the Callocate function 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. Line 9 prints out contents of myPointer which illustrates that the variable contains a memory address and not data. In line 11 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. Line 13 prints out the contents of the memory location which is now 10.

In line 15, 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 8.1: Output of 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 addressof 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 8.1: Pointer Operators

You will notice that the addressof 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

Function	Syntax	Comment
		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 size of 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 8.2: 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(datatype)`. 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 a way 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, but most programmers use the index method because of its simplicity.

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

```
Option Explicit
```

```
2
3   Dim myPtr As Integer Ptr
4   Dim As Integer i
5
6   'Try and allocate space for 10 integers
7   myPtr = Callocate(10, Sizeof(Integer))
8
9   'Make sure the space was allocated
10  If myPtr = 0 Then
11      Print "Could not allocate space for buffer."
12      End 1
13  End If
14
15  'Load data into the buffer
16  Print "Loading data, print data using *..."
17  For i = 0 To 9
18      *(myPtr + i) = i
19      Print "Index:";i;" data:";*(myPtr + i)
20  Next
21  Print
22
23  'Print data from buffer
24  Print "Show data using indexing..."
25  For i = 0 To 9
26      Print "Index:";i;" data:";myPtr[i]
27  Next
28
29  'Free the memory
30  Deallocate myPtr
31
32  Sleep
33  End
```

Listing 8.2: ptraccess.bas

`Analysis:` In lines 3 and 4 the working variables are declared. Space for 10 integers is created in Line 7 using the Callocate function. Lines 10 through 13 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.

Lines 17 through 20 load and print the memory segment with ten integers using the indirection operator and pointer arithmetic. Lines 25 through 27 print out the values using the index method. Notice that the index method is much more compact and easier to read. In line 30 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 8.2: Output of 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

Function	Syntax	Comment
		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 addressof operator @.
Varptr	myPtr = Varptr(variable)	This function works the same way as the addressof operator @.

Table 8.3: 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.

```
1    Option Explicit
2
3    'Declare our function to be used with pointer
4    Declare Function Power(number As Integer, pwr As Integer) As Integer
5
6    'Dim a function pointer
7    Dim FuncPtr As Function(x As Integer, y As Integer) As Integer
8
9    'Get the address of the function
10   FuncPtr = @Power
11
12   'Use the function pointer
13   Print "2 raised to the power of 4 is";FuncPtr(2, 4)
14
15   Sleep
16   End
17
18   'Write the function that will be called
19   Function Power(number As Integer, pwr As Integer) As Integer
20       Return number^pwr
21   End Function
```

Listing 8.3: funcptr.bas

`Analysis:` Line 4 declares the function prototype that will be used with the function pointer. Line 7 declares the function pointer using the 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. Line 10 initializes the function pointer to the address of the function using the address of operator @. You could use Procptr here as well. Line 13 uses 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. Lines 19 through 21 define the actual Power function.

Running the program will produce the following result.

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

Listing 8.4: Output of 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 in stdlib.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.

```
Option Explicit

#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))
```

```
56          el2 = *(Cptr(Integer Ptr, e2))
57          Print "Qsort called";cnt;" time(s) with";el1;" and";el2;"."
58          'Compare the values
59          If el1 < el2 Then
60              Return -1
61          Elseif el1 > el2 Then
62              Return 1
63          Else
64              Return 0
65          End If
66  End Function
```

Listing 8.5: qsort.bas

`Analysis:` Line 3 includes the crt.bi file 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. Line 7 declares 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. The undercores character _ at the end of line 7 is the line continuation character and tells the compiler that the following line is actually a part of this line. In line 11 an array of integers is dimensioned. The array will have 10 elements, with indexes from 0 to 9. Line 12 dimensions a working variable i, that will be used to load and display the array values.

Line 15 seeds 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. Lines 19 through 25 initializes the array with some random integers and displays them to the console screen. The code in line 22, 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 0 and 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 numbers will range from 0 to 19. In line 24 the value of the current array index is printed to the console screen.

In line 29 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 is 10, 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. Lines 34 through 37 print out the now sorted array. Lines 39 and 40 close the program in the usual way.

Lines 47 through 66 contain the compare function code. In line 47 the function is defined just like the declare statement in line 7 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. Lines 49 and 50 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.

Lines 55 and 56 use 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. Line 57 prints out the current call count and the passed parameter values.

Line 59 through 65 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 in line 66.

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

```
Unsorted
i =  0 value =  14
i =  1 value =  18
i =  2 value =  2
i =  3 value =  14
i =  4 value =  3
i =  5 value =  15
i =  6 value =  16
i =  7 value =  4
i =  8 value =  10
i =  9 value =  19

Qsort called 1 time(s) with 18 and 15.
Qsort called 2 time(s) with 19 and 15.
Qsort called 3 time(s) with 10 and 15.
Qsort called 4 time(s) with 2 and 15.
Qsort called 5 time(s) with 14 and 15.
Qsort called 6 time(s) with 3 and 15.
Qsort called 7 time(s) with 14 and 15.
Qsort called 8 time(s) with 16 and 15.
Qsort called 9 time(s) with 4 and 15.
Qsort called 10 time(s) with 16 and 15.
Qsort called 11 time(s) with 4 and 15.
Qsort called 12 time(s) with 18 and 16.
Qsort called 13 time(s) with 19 and 18.
Qsort called 14 time(s) with 18 and 16.
Qsort called 15 time(s) with 10 and 4.
Qsort called 16 time(s) with 2 and 10.
Qsort called 17 time(s) with 14 and 10.
Qsort called 18 time(s) with 3 and 14.
Qsort called 19 time(s) with 14 and 14.
Qsort called 20 time(s) with 10 and 4.
Qsort called 21 time(s) with 2 and 10.
Qsort called 22 time(s) with 14 and 10.
Qsort called 23 time(s) with 3 and 14.
Qsort called 24 time(s) with 10 and 4.
Qsort called 25 time(s) with 2 and 10.
Qsort called 26 time(s) with 3 and 10.
Qsort called 27 time(s) with 3 and 4.
Qsort called 28 time(s) with 2 and 4.
Qsort called 29 time(s) with 3 and 2.

Sorted
i =  0 value =  2
i =  1 value =  3
i =  2 value =  4
i =  3 value =  10
i =  4 value =  14
i =  5 value =  14
i =  6 value =  15
i =  7 value =  16
i =  8 value =  18
i =  9 value =  19
```

Output 8.3: 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.

```
1    Option Explicit
2
3    'Declare a pointer to an int pointer
4    Dim myMemArray As Integer Ptr Ptr
5    Dim As Integer i, j
6
7    'Create 10 rows of integer pointers
8    myMemArray = Callocate(10, Sizeof(Integer Ptr))
9
10   'Add 10 columns of integers to each row
11   For i = 0 To 9
12       myMemArray[i] = Callocate(10, Sizeof(Integer))
13   Next
14
15   'Add some data to the memory segment
16   For i = 0 To 9
17       For j = 0 To 9
18           myMemArray[i][j] = Int(Rnd * 10)
19       Next
20   Next
21
22   'Print out data
23   For i = 0 To 9
24       For j = 0 To 9
25           Print "i,j = ";i;",";j;" Mem Array =";myMemArray[i][j]
26       Next
27   Next
28
29   'Free memory segment
30   For i = 0 To 9
31       Deallocate myMemArray[i]
32   Next
33   'Free the pointer to pointer
34   Deallocate myMemArray
35
36   Sleep
37   End
```

Listing 8.6: memarray.bas

`Analysis:` Line 4 dimensions 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. Line 5 just declares some working variables that are used in the For-Next loops later in the program.

Line 8 creates the rows of the array by allocating a memory segment that will contain 4 integer pointers, which are initialized in lines 11 through 13. Remember that the index method of accessing a pointer automatically does the pointer arithmetic for you. The code in lines 11 through 13 iterates through the memory segment created in line 8 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.

Lines 16 through 20 add some random numbers to the memory array. Notice that by using the indexing method you can access the memory array just like a normal array. i points 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 in lines 23 through 27 to print out the array values.

Lines 30 through 32 free the individual rows of memory that were created in line 8. It is important that each row be freed before freeing myMemArray. If you were to just free myMemArray, the rows would still be in memory, but unaccessible, causing a memory leak. Once all the rows have been freed, myMemArray can be freed in line 34. 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. Lines 36 and 37 close the program 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 8.4: 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.

A Look Ahead

Pointers are an efficient and powerful data type, and you will be seeing more of them in this book, including the next chapter where you will explore the string data types available in FreeBasic.

9 String Data Types

Strings are an essential part of programming, and proper manipulation of strings is an essential skill for any programmer. You will need to use strings 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 are solutions 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 está fresco!“ 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 alpha-numeric 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 scancode 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 9.1: 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 36 byte blocks, which reduces the amount of resizing that the compiler must do if the string size changes. The following program displays the internal string structure data for a typical dynamic string[3].

```
Option Explicit

'String descriptor
Type StringDesc
  strdata As Zstring Ptr
  length As Integer
  size As Integer
End Type

'Create a string
Dim As String myString = "This is a dynamic string in FreeBasic!"
'Dimension a pointer to the string descriptor
```

[3]String descriptor code provided by FreeBasic forum member cha0s.

```
13   Dim As StringDesc Ptr StringData = Cptr(stringdesc Ptr, @myString)
14
15   'Print string descriptor data
16   Print "Data: ";*(StringData->strdata)
17   Print "Length: ";StringData->length
18   Print "Size: ";StringData->size
19
20   Sleep
21   End
```

Listing 9.1: dynastring.bas

Analysis: Lines 4 through 8 defines the string descriptor data type. The strdata field is a pointer to a NULL terminated string in memory. The length fields contains the number of characters in the string. The size field indicates the length of the current memory-block allocation. Line 11 creates a dynamic string and initializes the value of the string. Line 13 creates a pointer, StringData, that points to the string descriptor of myString. Notice that the address of operator is used to return the address of the string descriptor, which is then cast as a StringDesc Ptr using the Cptr function.

Lines 16 through 18 then print out the string descriptor data. The member operator, ->, is used with type definition pointers and is comparable to using *PtrName. The member operator will be discussed in the chapter on composite types. The program is closed in the usual way.

Running the program in FBIde will produce the following result.

```
Data: This is a dynamic string in FreeBasic!
Length:  38
Size:  72
```

Output 9.1: Output of dynastring.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 runtime 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.

```
1    Option Explicit
2
3    'Define a fixed length string
4    Dim myFString As String * 20
5
6    'Save some dat in the string
7    myFString = "This should fit."
8    Print myFString
9
10   'This data will be truncated.
11   myFString = "This string will be truncated."
12   Print myFString
13
14   Sleep
15   End
```

Listing 9.2: fixedstr.bas

`Analysis:` Lines 4 defines a fixed length string that can hold up to 20 characters. Line 7 loads the variable with a string and line 8 prints the contents of the variable. Since the string "This should fit." is smaller than 20 characters, so the whole string will be printed out. Line 11 tries to initialize the string with data that is longer than 20 characters so the string will be truncated when it is printed out in line 12.

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

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

Output 9.2: 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 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 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. You must also not include the character 0 in any of your data, or the data will be truncated since FreeBasic will see the Null character as the end of the string.

The following program shows both uses of the Zstring data type.

```
Option Explicit

'Declare a fixed length Zstring
Dim myFixedZstring As Zstring * 20 + 1
'Declare a dynamic Zstring
Dim myDynaZstring As Zstring Ptr

'Load some data into the fixed zstring
Print "123456789012345678901"
myFixedZstring = "This should fit."
Print myFixedZstring
myFixedZstring = "This will be truncated."
Print myFixedZstring

'Create a dynamic zstring
Print "123456789012345678901"
myDynaZstring = Callocate(20 + 1)
*myDynaZstring = "Let's add some data."
Print *myDynaZstring
'Resize the string: 20 for current data + 5 for new
'data + 1 for Null
myDynaZstring = Reallocate(myDynaZstring, 20 + 5 + 1)
Print "12345678901234567890123456"
```

```
24   *myDynaZstring = "Let's add some more data."
25   Print myDynaZstring[0]
26
27   'Deallocate the memory segment
28   Deallocate myDynaZstring
29
30   Sleep
31   End
```

Listing 9.3: zstring.bas

Analysis: Line 4 declares a fixed length Zstring. Notice that the desired data length is 20 so 1 is added to to allocate space for the terminating Null character. Line 6 declares a pointer to a Zstring that will be managed dynamically. Line 9 prints out a ruler to compare the print statements in line 11 and 13. Line 10 initializes the Zstring with some data that is small enough to fit in the declared size. Line 12 adds some data that is too long to fit within the Zstring, which will be truncated to fit.

Line 16 prints another ruler for comparison with the dynamic Zstring data. Line 17 Callocates enough space for a 20-byte string along with the terminating Null character. Since myDynaZstring is defined as a Zstring Ptr, the indirection operator is used to initialize the variable data in line 18. Line 22 resizes the Zstring to hold 5 additional characters plus the terminating Null using the Reallocate function. Another ruler is printed in line 23 to show the new size of the data loaded into the variable in line 24. Line 25 uses the pointer index method to print out the contents of the variable. Line 28 deallocates the allocated memory.

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

```
123456789012345678901
This should fit.
This will be truncat
123456789012345678901
Let's add some data.
12345678901234567890123456
Let's add some more data.
```

Output 9.3: Output of zstring.bas

As you can see, the fixed length version of the Zstring data type behaves exactly like the fixed string data type and the dynamic version behaves exactly like any other pointer-based memory segment. Since there is no intrinsic method for determining the size of a Zstring allocation, you must be careful not to over-run the buffer. The best way to do this is to keep a current allocation size in a separate variable that can be referenced as needed.

You will notice in Line 25 of the listing that the pointer index method was used to print the contents of the resized Zstring. What would the output be if you changed the index from 0 to 6? Remember that a pointer index behaves much like an array index, and a Zstring could be viewed as an array of characters. If you were to use 6 as the index, the output would be `add some more data`. The Print statement would start at character position 6 and print everything up to the terminating Null character.

You can directly assign a fixed length Zstring to a dynamic string. For a dynamic Zstring, the pointer must be dereferenced by using * or the pointer index method. Keep in mind that the behavior of the indexing method is the same for assignments as it is for the Print statement.

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 will be discussed in more detail in the chapter on Unicode.

String Functions

Creating and using strings in your application will often consist of manipulating those strings and FreeBasic has a rich set of string functions, which are listed in the following table.

Function	Syntax	Comment

Asc	B = Asc(string)
	B = Asc(string, position)
		Returns the character code of the first character in a string as a uninteger, or the character code of character at position.

Bin	B = Bin(number)
	B = Bin(number, digits)
		Returns the binary form of number as a string. Can optionally return only number of digits.

Chr	B = Chr(code)
	B = Chr(code, code, ...)
		Returns the character represented by Ascii Code. If multiple codes are passed to Chr, the function will return a string of characters.

Hex	B = Hex(number)
	B = Hex(number, digits)
		Returns the hexadecimal form of number as a string. Can optionally return only number of digits.

Instr	B = Instr(string, substring)
	B = Instr(start, string,
		Returns the position of substring within string as an

Function	Syntax	Comment
	substring)	integer. Will accept an optional start position. If substring is not found, 0 is returned.
Lcase	B = Lcase(string)	Converts string to all lowercase.
Left	B = Left(string, number)	Returns the leftmost number of characters from string.

Len	B = Len(string)
	B = Len(data_type)
		Returns the length of a string or the length of a numeric data type.

Lset	Lset(string_variable, string)	Left justifies string within string varibale.

Ltrim	B = Ltrim(string)
	B = Ltrim(string, trimset)
	B = Ltrim(string, ANY trimset)
		The first format will trim all spaces from left side of string. The second format will trim characters from left side of string if they exactly match trimset. The third format will trim characters from left side of string if they match any in trimset.

Mid (Function)	B = Mid(string, start)
	B = Mid(string, start, length)
		Returns a substring from string starting at start to the end of the string, or of length.

Mid (Statement)	Mid(string, start) = B Mid(string, start, length) = B	Copies contents of B into string starting at start for length. The current characters in string are replaced. If no length is given, all of B is inserted.

Oct	B = Oct(number)
	B = Oct(number, digits)
		Returns the octal form of number as a string. Can optionally return only number of digits.

Right	B = Right(string, number)	Returns the rightmost number of characters from string.
Rset	Rset(string_variable, string)	Right justifies string within string varibale.

Rtrim	B = Rtrim(string)
	B = Rtrim(string, trimset)
	B = Rtrim(string, ANY trimset)
		The first format will trim all spaces from right side of string. The second format will trim characters from right side of string if they

Function	Syntax	Comment
		exactly match trimset. The third format will trim characters from right side of string if they match any in trimset.
Space	B = Space(number)	Returns a string with number of spaces.

String	B = String(number, code)
	B = String(number, string)
		String will return a string with number of characters that correspond to the ascii character code or the first character of string.

Trim	B = Trim(string)
	B = Trim(string, trimset)
	B = Trim(string, ANY trimset)
		The first format will trim all spaces from left and right side of string. The second format will trim characters from left and right side of string if they exactly match trimset. The third format will trim characters from left and right side of string if they match any in trimset.

Ucase	B = Ucase(string)	Converts string to all uppercase.

Table 9.2: FreeBasic String Functions

Len Versus Sizeof

For numeric data types, Len and Sizeof both return the size of the data type. For string data types, Len returns the length of the string data and Sizeof returns the data type size. The following program illustrates the differences in the two functions.

```
Option Explicit

'declare some string variables
Dim myDynaString as String
Dim myFixedString as String * 20
Dim myZString as ZString * 30
Dim myWString as WString * 30

'add some data
myDynaString = "Hello World From FreeBasic!"
myFixedString = "Hello World!"
myZString = "Hello World From FreeBasic!"
myWString = "Hello World From FreeBasic!"
```

```
14
15  Print "Dynamic string: ";myDynaString
16  Print "Fixed string: ";myFixedString
17  Print "Zstring: ";myZString
18  Print "Wstring: ";myWString
19  Print
20
21  'print out the lengths of strings
22  Print "Length of Dynamic String is";Len(myDynaString);" byte(s)."
23  Print "Length of Fixed String";Len(myFixedString);" byte(s)."
24  Print "Length of ZString is";Len(myZString);" byte(s)."
25  Print "Length of WString is";Len(myWString);" byte(s)."
26  Print
27
28  'print out the variable size
29  Print "Size of Dynamic String is";SizeOf(myDynaString);" byte(s)."
30  Print "Size of Fixed String";SizeOf(myFixedString);" byte(s)."
31  Print "Size of ZString is";SizeOf(myZString);" byte(s)."
32  Print "Size of WString is";SizeOf(myWString);" byte(s)."
33
34  'wait for keypress
35  Sleep
36  End
```

Listing 9.4: strtype.bas

`Analysis:` Lines 4 through 7 dimension a string of each type. Lines 10 through 13 initialize the string variables with some string data. Lines 15 through 18 print out the string data for reference. Lines 22 through 25 use the Len function to print out the length of the actual string data. Lines 29 through 32 use the Sizeof function to print out the length of the data types. The program is closed in the usual manner.

Running the program should produce the following output.

```
Dynamic string: Hello World From FreeBasic!
Fixed string: Hello World!
Zstring: Hello World From FreeBasic!
Wstring: Hello World From FreeBasic!

Length of Dynamic String is 27 byte(s).
Length of Fixed String 20 byte(s).
Length of ZString is 27 byte(s).
Length of WString is 27 byte(s).
```

```
Size of Dynamic String is 12 byte(s).
Size of Fixed String 21 byte(s).
Size of ZString is 30 byte(s).
Size of WString is 60 byte(s).
```

Output 9.4: Output of stringtype.bas

As you can see from the output, Len and Sizeof return different values. The Len function automatically dereferences the string variables and returns the length of the actual string, while the Sizeof function returns the length of the string variable itself. For a dynamic string, Sizeof returns the length of the string descriptor. The string descriptor contains a pointer to a Zstring (4 bytes), an integer containing the length of the string (4 bytes) and an integer that contains the size of the current allocation (4 bytes) which total to 12 bytes. For fixed length string and for Zstrings, Sizeof returns the dimensioned size of the variable. For a Wstring, the size is twice the dimensioned value because a Wstring uses 16-bit characters rather than 8-bit characters. The Sizeof function is useful for determining the allocation size of fixed length strings, fixed allocation Zstrings and fixed allocation Wstrings so that you do not inadvertently lose data by trying to initialize the variable with more data than it can hold. Dynamic strings can contain variable length data so Sizeof is rarely used with dynamic strings.

Using String Functions with Zstring Pointers

There will be times when you will need to use a string function with a Zstring pointer. If you try to use the pointer with directly with a string function you will receive a Type Mismatch error from the compiler. What you need to do is to dereference the pointer when passing the Zstring to a string function so that the function can access the string data directly. The following program illustrates this concept.

```
1    Option Explicit
2
3    Dim myZstring As Zstring Ptr
4
5    myZstring = Callocate(10, Sizeof(Byte))
6    *myZstring = "Hello FB!"
7
8    Print "F is at position";Instr(*myZstring, "F")
9
10   Deallocate myZstring
11
12   Sleep
13   End
```

Listing 9.5: zstringfunc.bas

`Analysis:` Line 3 dimensions a Zstring pointer, myZstring. Line 5 allocates some space for the string data in line 6. Line 8 uses the Instr function to determine the position of the F character in the string and prints the position to the screen. Notice that the dereference operator is used inside the function with myZstring so that Instr has access to the string data. Line 10 deallocates the memory assigned to the variable. The program is then closed in the usual way.

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

```
F is at position 7
```

Output 9.5: Output of zstringfunc.bas

This technique only applies to a Zstring pointer. A fixed length Zstring can be passed directly to the string functions with no need to dereference the variable.

The MK* and CV* String Functions

There are times when you will want to save numeric data to a disk file, such as when you are creating your own database system. Saving numeric data as strings can be problematic since the string representation of the data can vary. Once solution to this problem is to use the various MK* functions which convert the binary representation of a number into a string, and CV* functions which convert the string back into a number. The advantage of using these functions is consistent numeric representation; an integer is converted into a 4-byte string, a double is converted into an 8-byte string. This makes saving and reading binary data from the disk quite easy. The following table lists the MK* and CV* functions.

Function	Syntax	Comment
Mkd	B = Mkd(number)	Converts a double-type number to a string with length of 8 bytes.
Mki	B = Mki(number)	Converts an integer-type number to a string with length of 4 bytes.
Mkl	B = Mkl(number)	Converts a long-type number to a string with length of 4 bytes.
Mklongint	B = Mklongint	Converts a longint-type number to a string with length of 8 bytes.
Mks	B = Mks(number)	Converts a single-type number to a string with length of 4 bytes.
Mkshort	B = Mkshort	Converts a short-type number to a string with length of 2 bytes.
Cvd	B = Cvd(string)	Converts an 8 byte string

Function	Syntax	Comment
		created with Mkd into a double-type number.
Cvi	B = Cvi(string)	Converts a 4 byte string created with Mki into an integer-type number.
Cvl	B = Cvl(sring)	Converts a 4 byte string created with Mkl into an integer-type number.
Cvlongint	B = Cvlongint(string)	Converts an 8 byte string created with Mklongint into a longint-type number.
Cvs	B = Cvs(string)	Converts a 4 byte string created with Mks into a single-type number.
Cvshort	B = Cvshort(string)	Converts a 2 byte string created with Mkshort into a short-type number.

Table 9.3: MK* and CV* String Conversion Functions

The following program shows how to use the functions.

```
Option Explicit

'Create some numeric variables
Dim As Integer myInt, myIntC, i
Dim As Double myDbl, myDblC
'Create some string variables
Dim As String mySInt, mySDbl

'Load some data
myInt = 10
myDbl = 254.56
Print "Integer: ";myInt
Print "Double: ";myDbl
Print
'Convert to strings
mySInt = Mki(myInt)
mySDbl = Mkd(myDbl)
Print
'Print out values
Print "Mki: ";
For i = 1 To Len(mySInt)
    Print Asc(Mid(mySInt, i, 1));" ";
```

```
23  Next
24  Print
25  Print "Mkd: ";
26  For i = 1 To Len(mySDbl)
27      Print Asc(Mid(mySDbl, i, 1));" ";
28  Next
29  Print
30  'Convert back to numbers
31  myIntC = Cvi(mySInt)
32  myDblC = Cvd(mySDbl)
33  Print
34  Print "Cvi: ";myIntC
35  Print "Cvd: ";myDblC
36
37  Sleep
38  End
```

Listing 9.6: mkcv.bas

`Analysis:` Line 4 creates three integer-type variables, myInt which will contain the initial integer value, myIntC which will contain the numeric value after the Cvi conversion and i which is a variable that will be used in the following For loop. Line 5 create two double-type variables, myDbl which will contain the initial double value and myDblC which will contain the Cvd conversion value. Line 7 creates two string variables, mySInt and mySDbl which will contain the Mki and Mkd string conversion values respectively.

Lines 10 and 11 set the initial values of the variables and lines 12 and 13 print out those values for reference. Lines 16 and 17 convert the integer and double values to binary string representations using the Mki and Mkd functions. Line 21 through 23 print out the ascii code values for the converted integer string. The Len function is used to determine the length of the string, which in the case of the integer will be four bytes. The ascii codes are print rather than the characters since the string is a binary representation of the integer value and some of the characters will not be visible on the screen. Lines 26 through 28 print out the character values for the converted double-type value.

Lines 31 and 32 convert the string representations back into numeric values which are printed to the screen in lines 34 and 35.

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

```
Integer:  10
Double:  254.56

Mki: 10 0 0 0
Mkd: 82 184 30 133 235 209 111 64
```

```
Cvi:  10
Cvd:  254.56
```

Output 9.6: Output of mkcv.bas

You can see from the output that the Mk* functions create an exact binary representation of the numeric values. Since the Mk* functions create strings that are the same length as their numeric counterparts, you have a very consistent representation of the numbers. This makes these functions quite useful when creating variable length records that will be stored on the disk.

If your program requires fixed-length records, you can use a Type definition to create disk records. This technique will be covered in the chapter on file handling.

Numeric String Conversion Functions

You find as you write programs that there are instances where you need to convert a text string such as "124.5" into a number, and the number 124.5 into a string. FreeBasic has several conversion functions that can be used to accomplish these tasks.

Function	Syntax	Comment
Format	B = Format(number, format_string)	Returns a formatted number. You must include "`vbcompat.bi`" in your program to use this function.
Str	B = Str(number)	Converts a numeric expression to a string representation. That is, 145 will become "145".
Val	B = Val(string)	Converts a string to a double value. The Val functions will convert from left to right, ending at the first non-numeric character.
Valint	B = Valint(string)	Converts a string to an integer value.
Vallng	B = Vallng(string)	Converts a string to a long integer value.
Valuint	B = Valuint(string)	Converts a string to a unsigned integer value.
Valulong	B = Valulong(string)	Converts a string to an unsigned long integer value.

Table 9.4: Numeric String Functions

These functions work just like the Mk* and Cv* functions, except that these functions work with text representations of the numeric values rather than binary representations. A common usage of these functions is in reading text files that contain text numbers, such as ini files, and the text needs to be converted to a number, and then back to a string for output to the disk. They are also useful for getting input from the user, a technique you will see in the chapter on the console functions.

The Format function is a general purpose function that can format numbers, dates and time. A seperate chapter is devoted to using the Format function.

Wide String Functions

Since wide strings contain 16-bit characters, there are a few string functions that work specifically with wide strings. The functions listed in the following table behave in the same manner as their 8-bit counterparts.

Function	Syntax	Comment

Wbin	B = Wbin(number)
	B = Wbin(number, digits)
		Returns the binary form of number as a wide string. Can optionally return only number of digits.

Wchr	B = Wchr(unicode)
	B = Wchr(unicode, unicode, ...)
		Returns the character represented by Unicode. If multiple codes are passed to Wchr, the function will return a string of unicode characters.

Whex	B = Whex(number)
	B = Whex(number, digits)
		Returns the hexadecimal form of number as a wide string. Can optionally return only number of digits.

Woct	B = Woct(number)
	B = Woct(number, digits)
		Returns the octal form of number as a wide string. Can optionally return only number of digits.

Wspace	B = Wspace(number)	Returns a wide string with number of spaces.

Wstr	B = Wstr(number)
	B = Wstr(ascii_string)
		The first form of Wstr will return a wide string respestation of a number. The second form will convert an ascii string to a Unicode string.

Wstring	B = Wstring(number, code)
	B = Wstring(number, string)
		String will return a wide string with number of characters that correspond to the ascii character code or the first character of string.

Table 9.5: Wide String Functions

The wide string functions work in the same manner as their regular string counterparts.

String Operators

There are two string operators `&` and `+` which concatenate two or more strings together. `&` is preferred over `+` since `&` will automatically convert the operands to strings, where `+` will not.

CRT Character and String Functions

The C Runtime Library has a number of functions to identify different characters and manipulate strings. These functions are useful for managing string data such as identifying numbers or parsing tokens within a string. While all of these actions can be coded using Basic code, using the CRT functions can speed development time, and in some cases operate much faster than equivalent Basic code.

Character Functions

In the ctype.bi declaration file, located in the inc\crt folder, you will find a number of functions to identify different types of characters. To use these functions in your program you must include the ctype.bi file, `#include once "crt/ctype.bi"`. The following table lists the different character functions. All of these functions take an ascii character code and return a non-zero result if successful or a zero result if not successful.

Function	Syntax	Comment
Isalnum	B = Isalnum(asc_code)	Returns non-zero if the character is an alpha-numeric character, zero if not.
Isalpha	B = Isalpha(asc_code)	Returns non-zero if the character is an alphabetical character, zero if not.
Iscntrl	B = Iscntrl(asc_code)	Returns non-zero if the character is a control character, such as a Tab, zero if not.
Isdigit	B = Isdigit(asc_code)	Returns non-zero if the character is a numeric digit, 0 to 9, zero if not.
Isgraph	B = Isgraph(asc_code)	Returns non-zero if the characterhas a glyph associated with it, zero if not. This function only identifies the standard glyph characters in the lower ascii range.

Function	Syntax	Comment
Islower	B = Islower(asc_code)	Returns non-zero if the character is lower-case, zero if not.
Isprint	B = Isprint(asc_code)	Returns non-zero if the character is a printable character, zero if not. This function only identifies the printable characters in the lower ascii range.
Ispunct	B = Ispunct(asc_code)	Returns non-zero if the character is a punctuation character, zero if not.
Isspace	B = Isspace(asc_code)	Returns non-zero if the character is a whitespace character, zero if not. Whitespace is defined as a space, form-feed, new-line, carriage return, horizontal and vertical tabs.
Isupper	B = Isupper(asc_code)	Returns non-zero if the character is upper-case, zero if not.
Isxdigit	B = isxdigit(asc_code)	Returns non-zero if the character is a hexidecimal digit, zero if not. Hexadecimal digits include 0 through 9, A through F and a through f.

Table 9.6: CRT Character Functions

The following program illustrates how to use these functions.

```
Option Explicit

#include once "crt/ctype.bi"

Dim As String myString
Dim As Integer ret, i, ichar

'Load some characters into the string
myString = "ABcd01!*  " & Chr(128) & Chr(1)

'Print header on console
Print "Char";Tab(6);"Alnum";Tab(12);"Alpha";Tab(18);"Cntrl";Tab(24);
Print "Digit";Tab(30);"Graph";Tab(36);"Lower";Tab(42);"Print";Tab(48);
Print "Punct";Tab(54);"Space";Tab(60);"Upper"
Print String(79, "-")

'Examine each character in string.
For i = 1 To Len(myString)
    ichar = Asc(Mid(myString, i, 1))
    'Print the character
    Print Chr(ichar);
    Print Tab(6);
    'Check to see what kind of character it is
    ret = isalnum(ichar)
    'If character type then print Y
    If ret <> 0 Then
        Print "Y";
    Else
        Print "N";
    End If
    Print Tab(12);
    ret = isalpha(ichar)
    If ret <> 0 Then
        Print "Y";
    Else
        Print "N";
    End If
    Print Tab(18);
    ret = iscntrl(ichar)
    If ret <> 0 Then
        Print "Y";
    Else
        Print "N";
    End If
    Print Tab(24);
    ret = isdigit(ichar)
    If ret <> 0 Then
        Print "Y";
    Else
        Print "N";
    End If
    Print Tab(30);
    ret = isgraph(ichar)
    If ret <> 0 Then
        Print "Y";
    Else
        Print "N";
```

```
58       End If
59       Print Tab(36);
60       ret = islower(ichar)
61       If ret <> 0 Then
62           Print "Y";
63       Else
64           Print "N";
65       End If
66       Print Tab(42);
67       ret = isprint(ichar)
68       If ret <> 0 Then
69           Print "Y";
70       Else
71           Print "N";
72       End If
73       Print Tab(48);
74       ret = ispunct(ichar)
75       If ret <> 0 Then
76           Print "Y";
77       Else
78           Print "N";
79       End If
80       Print Tab(54);
81       ret = isspace(ichar)
82       If ret <> 0 Then
83           Print "Y";
84       Else
85           Print "N";
86       End If
87       Print Tab(60);
88       ret = isupper(ichar)
89       If ret <> 0 Then
90           Print "Y"
91       Else
92           Print "N"
93       End If
94   Next
95
96   Sleep
97   End
```

Listing 9.7: crt_char.bas

`Analysis:` Line 3 includes the ctype.bi file that is needed to link to these CRT functions. Lines 5 and 6 declare the working variables. MyString will contain the characters to identify, ret is the return value from the function, i is used in the For-Next loop and ichar contains the fascicle code of the character that is passed as the parameter to the different functions. Line 12 through 15 display a header line that will be used to identify which functions returns a true result. The different column headers refer to the different functions. Alnum is the return column for the Isalnum function, the Alpha column is the return for the Isalpha function and so on.

Line 18 and 94 comprise the For-Next block. Line 19 gets the current character from the string and converts it to an ascii code using the Asc function. The character is then tested with each function. If the function returns a non-zero result, a Y is printed in the appropriate column. If the function returns a zero, an N is printed in the appropriate column.

The program is then closed in the usual way.

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

```
Char Alnum Alpha Cntrl Digit Graph Lower Print Punct Space Upper
----------------------------------------------------------------
A    Y     Y     N     N     Y     N     Y     N     N     Y
B    Y     Y     N     N     Y     N     Y     N     N     Y
c    Y     Y     N     N     Y     Y     Y     N     N     N
d    Y     Y     N     N     Y     Y     Y     N     N     N
0    Y     N     N     Y     Y     N     Y     N     N     N
1    Y     N     N     Y     Y     N     Y     N     N     N
!    N     N     N     N     Y     N     Y     Y     N     N
*    N     N     N     N     Y     N     Y     Y     N     N
     N     N     N     N     N     N     Y     N     Y     N
     N     N     N     N     N     N     Y     N     Y     N
Ç    N     N     N     N     N     N     N     N     N     N
☺    N     N     Y     N     N     N     N     N     N     N
```

Output 9.7: Output of crt_char.bas

As you can see, these functions work with the lower ascii characters, which are the characters you would normally find in text files. Using these functions can make your job a lot easier when you are trying to identify characters from a file or from user input.

Strtok Function

The Strtok function will return tokens separated by a delimiter set. Strtok is quite fast and since it will look for a set of delimiters, it is much easier to use Strtok than Instr within a loop. Strtok is contained in the string.bi declaration file located in the inc\crt folder of your FreeBasic installation. Strtok is declared as:

```
declare function strtok cdecl alias "strtok" (byval as zstring ptr, _
                                               byval as zstring ptr) as zstring ptr
```

The first parameter is the string to parse and the second parameter is the delimiter set. Strtok returns a pointer to the first token in the string. Even though the parameters are defined as Zstring pointers, you can use dynamic strings since the compiler will automatically dereference a dynamic string if it is passed to a Zstring pointer. This makes working with this function quite easy. You will need to define a Zstring pointer as the return value though.

The following program uses Strtok to parse a string into tokens.

```
1    Option Explicit
2
3    #include once "crt.bi"
4
5    Dim As String tstr, tmpstr, delim
6    Dim zret As Zstring Ptr
7
8
9    'Create delimiters
10   delim = " ,!-"
11   'Create parse string
12   tstr = "Hello-World, From Freebasic!"
13   'Create a working copy of string
14   'strtok will alter original string
15   tmpstr = tstr
16   'First call with string and delimiters
17   zret = strtok(tmpstr, delim)
18   'Check for a NULL pointer
19   If zret <> NULL Then
20       Print zret[0]
21       'Parse rest of string
22       Do
23           'Call with NULL to work on same string
24           zret = strtok(NULL, delim)
25           If zret <> NULL Then
26               Print zret[0]
27           End If
28       Loop Until zret = NULL
29   End If
30   Sleep
31   End
```

Listing 9.8: strtok.bas

`Analysis:` Line 3 includes the crt library declarations so that Strtok is available to the program. Lines 5 and 6 define the working variables. Tstr is the original parse string, tmpstr is a copy of the original string and delim is the set of delimiters. Zret is defined as a Zstring pointer and will be the return value from the function. Line 10 sets the delimiter set and line 12 sets the parse string. In line 15 a copy is made of the original string, since Strtok will alter the string as it is parsed. If you need to save the original string value, use a temporary string with the function rather than the original string. Line 17 calls Strtok with the parse string and the delimiter set. This initial call sets up the function to parse the string. Subsequent calls to Strtok will use a Null in place of the parse string to indicate that you want to parse the original string and not a new string.

Line 19 checks for a Null pointer. Strtok will return a Null when it cannot extract any tokens from the parse string. You should always check for a NULL pointer before using the pointer reference. Line 20 prints the value of the token using the pointer index method. This will print everything from index 0 up to the terminating Null character. Remember that Zstring are C-type strings, an array of characters that terminate with character zero, a Null.

Line 22 through 28 call Strtok with a Null for the parse string to extract each token from the string. Thre loop terminates when a Null is returned from Strtok. Once again, in line 25, the pointer is checked to make sure it isn't a Null pointer before the value is printed to the console window.

After all the tokens have been processed, zret will be Null, the loop will exit and the program will close in the usual manner.

When you run the program you should the following output.

```
Hello
World
From
Freebasic
```

Output 9.8: Output of strtok.bas

The output shows the individual tokens in the string. Not only is Strtok easy to use, but because you can pass a set of delimiters the amount of code you have to write to parse a string that has several delimiters is considerably less than if you wrote the parse function in FreeBasic. Less code means less chance of things going wrong, which results in a more robust, stable program.

A Look Ahead

So far in the book you have seen the individual intrinsic data types. There are times though when you need an aggregate data type to fully describe your data. This is where Type definitions and Unions come in handy, which are discussed in the next chapter.

10 Composite Data Types

There are times when creating a program that you may want to define an aggregate structure such as a personnel record, or an enemy in a game. While you can do this using individual data types, it is hard to manage within a program. FreeBasic offers two composite data types, the Type and Union.

Types

FreeBasic allows you to group several data types into a unified structure called a Type definition which you can use to describe these aggregate data structures.

The basic structure of a type definition is:

```
Type typename
        var definition
        var definition
        ...
End Type
```

The Type-End Type block defines the scope of the definition. You define the elements of the type structure in the same manner as using the Dim keyword, without using Dim. The following code snippet shows how to build an employee type.

```
Type EmployeeType
        fname As String * 10
        lname As String * 10
        empid As Integer
        dept As Integer
End Type
```

You can use any of the supported data types as data elements, including pointers and other type definitions. When you create the type definition, such as the example above, you are just creating a template for the compiler.   In order to use the type definition, you need to create a variable of the type, as the following code snippet illustrates.

```
Dim Employee As EmployeeType
```

Once you have created a variable of the type, you can access each element within the type using the dot notation `var_name.field_name`. Using the above example, to access the fname field you would use `Employee.fname = “Susan”`.

To access multiple fields at a time, you can use the With-End With block. The following code snippet shows how to use the With block with the above example.

```
With Employee
       .fname = "Susan"
       .lname = "Jones"
       .empid = 1001
       .dept = 24
End With
```

The following program shows how to define, create and manage a type definition.

```
1    Option Explicit
2
3    'Create type definition
4    Type EmployeeType
5        fname As String * 10
6        lname As String * 10
7        empid As Integer
8        dept As Integer
9    End Type
10
11   'Create an instance of the type
12   Dim Employee As EmployeeType
13
14   'Initialize the type
15   With Employee
16       .fname = "Susan"
17       .lname = "Jones"
18       .empid = 1001
19       .dept = 24
20   End With
21
22   'Print out header row
23   Print "First Name";Tab(13);"Last Name";Tab(25);"Emp ID";Tab(33);"Dept"
24   Print String(79, "-")
25   'Print out data
26   With Employee
27       Print RTrim(.fname);Tab(13);RTrim(.lname);Tab(24);.empid;Tab(32);.dept
28   End With
29
30   Sleep
31   End
```

Listing 10.1: type.bas

`Analysis:` Line 4 through 9 define the type structure that is used in the program. The type has 4 fields, two fixed length strings and 2 integer values. Dynamic strings can be used within a type definition, however if you want to save the type information to the disk, then you need to use fixed length strings. Dynamic strings are actually pointers to a string descriptor and saving a type that contains dynamic strings will save the 4 byte pointer value, rather than the actual string data, resulting in data loss.

Line 12 creates a variable of the type, Employee. The type definition is a template and cannot be used until you create an instance of the type by creating a variable of the type. Lines 15 through 20 initialize the type variable with some data using the a With-End With block.

Line 23 prints a header row to the console that indicates the field name data. The Tb function is used to align data names to the appropriate column. Line 24 uses the String function to print a dashed line, just to offset the header row from the data row. Lines 26 through 28 prints the type data. Rtrim is used on the fixed length string elements to trim off any unused trailing spaces. The Tab function is again used to align the data to the appropriate columns. The program is then ended in the usual way.

When you run the program you should the following output.

```
First Name   Last Name    Emp ID  Dept
-------------------------------------
Susan        Jones        1001    24
```

Output 10.1: Output of type.bas

As you can see from the program, using a type definition is a perfect way to group related data into a single data structure. Not only is it a compact way to describe data in your program, but by grouping related data into a single object, you can manipulate that data as a single entity, rather than as a bunch of unrelated variables. This reduces the chances that errors will creep into your program by trying to manage large a set of individual variables.

Types Within Types

In addition to the intrinsic data types, type fields can also be based on a type definition. Why would you want to do this? One reason is data abstraction. The more general your data structures, the more you can reuse the code in other parts of your program. The less code you have to write, the less chance of errors finding their way into your program. Using the Employee example, suppose for a moment that you needed to track more dept information than just the department id. You might need to keep track of the department manager, the location of the department, such as the floor or the building, or the main telephone number of the department. By putting this information into a separate type definition, you could this information by itself, or as part of another type definition such as the Employee type. By generalizing your data structures, your program will be smaller, and much more robust.

Using a type within a type is the same as using on of the intrinsic data types. The following code snippets illustrates an expanded department type and an updated employee type.

```
        Type DepartmentType
        id As integer
        managerid as integer
        floor as integer
End Type

Type EmployeeType
        fname As String * 10
        lname As String * 10
        empid As Integer
        dept As DepartmentType
End Type
Dim Employee As EmployeeType
```

To access the department information within the Employee type, you use the compound dot notation to access the dept fields.

```
Employee.dept.id = 24
Employee.dept.managerid = 1012
Employee.dept.floor = 13
```

The top levels is Employee, so that reference comes first. Since dept is now a type definition, you need to use the dept identifier to access the individual fields within the DepartmentType. You can even carry this further, by including a type within a type within a type. You would simply use the dot notation of the additional type level as needed. While there is no limit on the levels of nested type definitions, it gets to be a bit unwieldy when used with several levels.

You can also use the With-End With block with nested types, by nesting the With block, as illustrated in the following code snippet.

```
With Employee
        .fname = "Susan"
        .lname = "Jones"
        .empid = 1001
        With .dept
                .id = 24
                .managerid = 1012
                .floor = 13
        End With
End With
```

Notice that the second With uses the dot notation, .dept, to specify the next level of type definitions. When using nested With blocks, be sure that match all the End With statements with their correct With statements to avoid a compile error.

The following program is a modified version of the previous program illustrating the new type definitions.

```
Option Explicit

'Create the type definition
Type DepartmentType
       id As Integer
       managerid As Integer
       floor As Integer
End Type

Type EmployeeType
    fname As String * 10
    lname As String * 10
    empid As Integer
    dept As DepartmentType
End Type

'Create an instance of the type
Dim Employee As EmployeeType

'Initialize the type
With Employee
    .fname = "Susan"
    .lname = "Jones"
    .empid = 1001
    With .dept
        .id = 24
        .managerid = 1012
        .floor = 13
    End With
End With

'Print out header row
Print "First Name";Tab(13);"Last Name";Tab(25);"Emp ID";Tab(33);
Print "Dept";Tab(39);"Manager";Tab(47);"Floor"
Print String(79, "-")
'Print out data
With Employee
    Print Rtrim(.fname);Tab(13);Rtrim(.lname);Tab(24);.empid;Tab(32);
    With .dept
        Print .id;Tab(38);.managerid;Tab(46);.floor
    End With
End With

Sleep
End
```

Listing 10.2: type-type.bas

`Analysis:` Lines 4 through 8 define the new type for the department. This is used in the employee type definition is line 14. The field dept is defined as DepartmentType, in the same manner as Employee is defined in line 18, with the Dim of course. Line 21 through 30 use two With-End With blocks to initialize the type data. Employee is the first level type, while .dept, using the dot notation, is the second level type. It is important that you use the dot notation with the second level With block so that the compiler knows that you are referring to a type element within Employee. Lines 33 and 34 prints the header row with the additional department field information. Lines 37 through 42 then print the type data, again using a nested With block. Notice how using the With blocks document the type structure without needing any additional comments. While this isn't the primary reason to use a With block, it does create code that is easily understood. The program is closed in the usual way.

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

```
First Name  Last Name   Emp ID  Dept  Manager Floor
-----------------------------------------------------
Susan       Jones       1001    24    1012    13
```

Output 10.2: Output of type-type.bas

While it may not be readily apparent from the example, abstracting the data in this manner gives you a tremendous amount of flexibility in your program. A company will usually have more than one department, so by abstracting the department information into a separate type definition, you can create functions that manage the department information, while at the same time minimizing the impact on the employee data structure.

Type Assignments

Extending the idea of data abstraction further, it would be nice to be able to separate the initialization of the department type from the initialization of the employee type. By separating the two functions, you can easily add additional departments as needed. This is where you can use type assignments. Just as you can assign one intrinsic data type to another, you can assign one type variable to another type variable, providing they share the same type definition.

The following program abstracts the department initialization function and assigns the result to the department type within the Employee type.

```
Option Explicit

'Create the type definition
Type DepartmentType
       id As Integer
       managerid As Integer
       floor As Integer
End Type

Type EmployeeType
    fname As String * 10
    lname As String * 10
    empid As Integer
    dept As DepartmentType
End Type

'This function will init the dept type and return it to caller
Function InitDept(deptid As Integer) As DepartmentType
    Dim tmpDpt As DepartmentType

    Select Case deptid
        Case 24 'dept 24
            With tmpDpt
             .id = deptid
             .managerid = 1012
             .floor = 13
            End With
        Case 48 'dept 48
            With tmpDpt
             .id = deptid
             .managerid = 1024
             .floor  = 12
            End With
        Case Else 'In case a bad department id was passed
            With tmpDpt
             .id = 0
             .managerid  = 0
             .floor  = 0
            End With
    End Select
    'Return the dept info
    Return tmpDpt
End Function

'Create an instance of the type
Dim Employee As EmployeeType

'Initialize the Employee type
With Employee
    .fname = "Susan"
    .lname = "Jones"
```

```
52      .empid = 1001
53      .dept = InitDept(24) 'get dept info
54  End With
55
56  'Print out header row
57  Print "First Name";Tab(13);"Last Name";Tab(25);"Emp ID";Tab(33);
58  Print "Dept";Tab(39);"Manager";Tab(47);"Floor"
59  Print String(79, "-")
60  'Print out data
61  With Employee
62      Print Rtrim(.fname);Tab(13);Rtrim(.lname);Tab(24);.empid;Tab(32);
63      With .dept
64          Print .id;Tab(38);.managerid;Tab(46);.floor
65      End With
66  End With
67
68  Sleep
69  End
```

Listing 10.3: type-assign.bas

`Analysis:` This program is identical to the previous program, with the addition of the department initialization function in lines 18 through 43. The functions is defined in line 18 as returning the DepartmentType, with one parameter, the department id. Line 19 creates a temporary department type variable that will be initialized with the appropriate data, and returned from the function. The function uses a Select Case block to set the data within the temporary type. The Select statement will execute the block of code that matches the deptid. If no matches are found, the Else case will return a zero for each element. Since it is probable that no dept will have an id of 0, this value can be checked to make sure that the passed deptid is valid. The Return statement in line 42 returns the initialized dept information to the caller.

The function is used in line 53 to set the data values for the department in the Employee type. Although this program doesn't include a check for an invalid department id for clarity, you could add a check on the department id following the Employee initialization code to make sure that a valid department id was used in the function.

The rest of the program is the same as the previous program, and prints the employee information to the screen.

When you run the program, you should see that the output is identical to the previous program output.

```
First Name  Last Name   Emp ID  Dept  Manager Floor
----------------------------------------------------
Susan       Jones       1001    24    1012    13
```

Output 10.3: Output of type-assign.bas

By just adding a simple function to the program, you have made the program easier to maintain than the previous versions. If a new department is created, you can simply update the InitDept function with the new department information, recompile and the program is ready to go.

Pointers to Types

You can create a pointer to a type, just as you can create a pointer to any of the intrinsic data types, and the same rules apply. As with any pointer, you must allocate some memory for the pointer, dereference the data elements, and deallocate the pointer when you are done using it. Type pointers use the arrow notation, ->, rather than the dot notation to access the individual fields with a type.

In the previous program listing, the error checking on an invalid department id was clunky at best. A better method is to have the InitDept function return a success code that indicates a valid department id. The following program implements this strategy, and uses a DepartmentType pointer to hold the department information for the Employee type.

```
Option Explicit

#define deptok 0

'Create the type definition
Type DepartmentType
      id As Integer
      managerid As Integer
      floor As Integer
End Type

Type EmployeeType
    fname As String * 10
    lname As String * 10
    empid As Integer
    dept As DepartmentType
End Type

'This function will init the dept type and return it to caller
Function InitDept(deptid As Integer, dpt As DepartmentType Ptr ) As Integer
    Dim ret As Integer = deptok

    Select Case deptid
        Case 24 'dept 24
             dpt->id = deptid
             dpt->managerid = 1012
             dpt->floor = 13
        Case 48 'dept 48
```

```
            dpt->id = deptid
            dpt->managerid = 1024
            dpt->floor  = 12
        Case Else 'In case a bad department id was passed
            dpt->id = 0
            dpt->managerid  = 0
            dpt->floor  = 0
            ret = Not deptok
    End Select
    'Return the dept status
    Return ret
End Function

'Create an instance of the types
Dim Employee As EmployeeType
Dim tmpDept As DepartmentType Ptr

'Initialize the pointer
tmpDept = Callocate(Sizeof(DepartmentType))
'Get the department info, check return
If InitDept(24, tmpDept) <> deptok Then
    'Error on dept type
    Deallocate tmpDept
    Print "Invalid Department ID."
Else
    'Department ok, init the Employee type
    With Employee
        .fname = "Susan"
        .lname = "Jones"
        .empid = 1001
        .dept = *tmpDept 'Dereference the pointer
    End With
    'Don't need the dept info now
    Deallocate tmpDept
    'Print out header row
    Print "First Name";Tab(13);"Last Name";Tab(25);"Emp ID";Tab(33);
    Print "Dept";Tab(39);"Manager";Tab(47);"Floor"
    Print String(79, "-")
    'Print out data
    With Employee
        Print Rtrim(.fname);Tab(13);Rtrim(.lname);Tab(24);.empid;Tab(32);
        With .dept
            Print .id;Tab(38);.managerid;Tab(46);.floor
        End With
    End With
```

```
74   End If
75
76   Sleep
77   End
```

Listing 10.4: type-ptr.bas

Analysis: This version of the program has a few modifications from the previous versions, but overall it is the same program. Line 3 defines a return for the InitDept function. If the functions returns this code, the department id is valid; if the function does not return this code, the department id is invalid. The type definitions are the same as used previously, but the function in line 20 has changed.

The function now returns an integer value, and there are two parameters, a department id and a pointer to the DepartmentType. Line 21 dimensions and initializes the function return value to deptok. If there are not errors, this value will be returned. If the deptid is invalid, the Case Else will set the return value to Not deptok (-1) in line 36. This is a good technique to return boolean values from a function. By initializing the return variable to the “ok” status, you only need to set the variable to the “not ok” status, if an error occurs. Less typing, less code and fewer chances of introducing bugs into the program.

In the individual Case blocks, the arrow notation is used to access the individual fields of the type pointer. The -> automatically dereferences the pointer for you, making it easier to manipulate the individual fields within the type. Line 39 returns the success code of the function. You don't need to return any type information, because the function is actually updating the external department type variable through the pointer.

Line 44 dimensions a DepartmentType pointer. Which is initialized in line 47. Remember that Callocate allocates a memory segment, clears the segment and then returns the address of the segment which is stored in the variable tmpDept. In line 49 the InitDept function is called inside an If statement. If the function fails, that is returns Not deptok, then the program deallocates the pointer and prints an error message. If the function succeeds, that is returns deptok, then the Employee type is initialized and the data is printed to the screen. Notice in line 59 that the program is still using a type assignment, but because tmpDept is a pointer, the dereference operator must be used.

Line 62 shows the power of using pointers as intermediate data structures. Once the department data is in the Employee type, you don't need the department type reference. Line 62 deallocates tmpDept, freeing the memory it was using. There is no sense in wasting memory on data structures that you only use for a very short time.

The rest of the program is identical to the previous programs.

When you run this program you should again see the same output as the previous programs.

```
First Name  Last Name   Emp ID  Dept  Manager Floor
----------------------------------------------------
Susan       Jones       1001    24    1012    13
```

Output 10.4: Output of type-ptr.bas

This change, like the previous change, was minor—most of the original program is in intact—but the program is now much more robust, easier to manage, and less prone to error than the original program. The fact that the output has not changed even though the program has, is a good indicator that the changes that were made were did not create unwanted side effects in the program. This is what data abstraction does; minimize negative impacts on existing code.

Type Memory Arrays

In the chapter on pointers, you saw how to create a memory array of integers using pointers. You can do the same with Type definitions, in much the same way. The following program creates a dynamic memory array using a type.

```
Option Explicit

'Declare type
Type myType
    id As Integer
End Type

'Declare a pointer to the type
Dim myTypePtr As myType Ptr
Dim As Integer i

'Create a memory array of types
myTypePtr = Callocate(5, Sizeof(myType))

'Add some data to the type
For i = 0 To 4
    myTypePtr[i].id = i
Next

'Print data
Print "Initial data:"
For i = 0 To 4
    Print myTypePtr[i].id
Next
Print

'Resize the array
myTypePtr = Reallocate(myTypePtr, 10)
```

```
29
30  'Add the new data
31  For i = 4 To 9
32      myTypePtr[i].id = i
33  Next
34
35  'Print data
36  Print "New Data:"
37  For i = 0 To 9
38      Print myTypePtr[i].id
39  Next
40
41  'Release memory
42  Deallocate myTypeptr
43
44  Sleep
45  End
```

Listing 10.5: memtype.bas

`Analysis:` Lines 4 through 6 declare a simple type. Line 9 dimensions a pointer to myType, and line 10 creates an integer variable i for the following For-Next loops. Line 13 allocates memory for 5 myType entries in the memory array. Lines 16 through 19 initialize the individual type elements. Notice the syntax is *type_variable*[index].*field_name*. Since myTypePtr is a typed pointer, you can use the index method to access each element in the array. You then use the dot notation to access each individual field of the type, just as you would using a single type variable. Lines 21 through 25 print the data to the screen using the same pointer index method.

Line 28 resizes the memory array with an additional 5 type elements. Lines 31 through 33 initialize the new memory segment with data, and lines 26 through 39 print the all the data to the screen. Line 42 deallocates the memory and the program is closed in the usual way.

When you run the program you should the following output.

```
Initial data:
 0
 1
 2
 3
 4

New Data:
```

```
0
1
2
3
4
5
6
7
8
9
```

Output 10.5: Output of memtype.bas

As you can see, creating a memory array of types is very straight forward. You create the number of elements you need by using Callocate and then using the index pointer method, you can access each individual type field using the dot notation. The advantage of using this method is that you can create dynamic structures in memory and grow or shrink them as necessary so that you are not wasting memory on elements you may not need. You can of course create a standard array of types, which will be covered in the chapter on arrays, and you can even create a dynamic array of types—but the method shown here does not incur the overhead of a dynamic array, and can be passed to functions using the pointer method.

Dynamic Arrays in Types

FreeBasic does allow you you to create an array within a type, which is covered in the chapter on arrays, but it does not allow you to create dynamic arrays in types. There are times when you will need to create a dynamic structure in a type, and this can be accomplished by using memory array method within a type. The following program illustrates a dynamic array of a type within a type.

```
Option Explicit

'This will be the dynamic type
Type pt
    row As Integer
    col As Integer
End Type

'The base type:
'darray will contain array data
Type DType
    darray As pt Ptr
End Type

'Create an instance of type
```

```
16  Dim myType As DType
17  Dim As Integer i
18
19  'Create enough space for elements
20  myType.darray = Callocate(5, Sizeof(pt))
21
22  'Load data into array
23  For i = 0 To 4
24      myType.darray[i].row = Int(Rnd * 10) + 1
25      myType.darray[i].col = Int(Rnd * 10) + 1
26  Next
27
28  'Print data
29  For i = 0 To 4
30      Locate myType.darray[i].row, myType.darray[i].col
31      Print "X"
32  Next
33
34  'Free allocated space
35  Deallocate myType.darray
36
37  Sleep
38  End
```

Listing 10.6: dynatype.bas

`Analysis:` Lines 4 through 7 define a type that will be used in the dynamic array. The program will use the row and column to print a character to the screen. Lines 11 through 13 define the base type. Notice that darray is dimensioned as a pointer to the type pt. If you needed an array of integers, you would use As Integer Ptr. The methods are the same whether using an intrinsic data type, or another type as illustrated here. Line 20 allocates memory for 5 array elements. Since the program is creating an array of types, the second parameter to Callocate is the size of the type, pt. For an integer array it would be `Sizeof(Integer)`. Lines 23 through 26 initialize the type array with a random row and column which will be used within the print code that follows. The access syntax in lines 24 and 25 follow what you have already seen; the dot notation is used to access the base type field darray, which is then accessed using the pointer indexing method, since the program is using a typed pointer. The dot notation is then used to access the individual fields of the array type. The syntax is the same as that shown with the type memory array, except here you have one additional level of indirection because the array is contained within a type.

Lines 29 through 32 use the row and column fields to print an X to the screen. Again you can see that the dot notation and pointer index methods are used to access the fields within the type. Line 35 frees the memory allocated and the program is closed in the usual way.

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

```
     X
        X

        X

    X


       X
```

Output 10.6: Output of dynatype.bas

The location of the X's will vary since the rows and columns are generated randomly, but since Randomize is not being used, you should the same layout between runs.

The two previous programs are quite similar in format. The differences are the levels of indirection being used. In the memory array program, the type itself was being used as the array element, so the pointer index was used at the top level, with the dot notation used to access the field data. In this program, the memory array is embedded within the type, so you use the standard dot notation to access the base type field, darray, and then use the pointer index method on the field to select the array element. Since the array element is a type, you use the dot notation to access the array element field, just as you would as if it were a simple type variable.

You could of course create a memory array that contained a memory array. In this case, you would use the pointer index method to select the base type element, and then the dot notation to select the memory array field, then a pointer index to access the element within the embedded array, and finally the dot notation to select the array element field. It sounds complicated, but is actually easier to implement than it is to explain. The following program modifies the previous program to illustrate this concept.

```
Option Explicit

'This will be the embedded type
Type pt
    row As Integer
    col As Integer
End Type

'The base type:
'darray will contain array data
Type DType
    darray As pt Ptr
```

```
13  End Type
14
15  'Create a pointer to base type
16  Dim myType As DType Ptr
17  Dim As Integer i, j
18
19  'Set up random number generator
20  Randomize Timer
21
22  'Create enough space for 3 base type elements
23  myType = Callocate(3, Sizeof(Dtype))
24  'Create space for 3 pt elements within base type
25  For i = 0 To 2
26      myType[i].darray = Callocate(3, Sizeof(pt))
27  Next
28
29  'Load data within the type arrays
30  For i = 0 To 2
31      For j = 0 To 2
32          myType[i].darray[j].row = Int(Rnd * 10) + 1
33          myType[i].darray[j].col = Int(Rnd * 10) + 1
34      Next
35  Next
36
37  'Print data
38  For i = 0 To 2
39      For j = 0 To 2
40          Locate myType[i].darray[j].row, myType[i].darray[j].col
41          Print "X"
42      Next
43  Next
44
45  'Free embedded type array
46  For i = 0 To 2
47      Deallocate myType[i].darray
48  Next
49  'Free base type array
50  Deallocate myType
51
52  Sleep
53  End
```

Listing 10.7: dyna-type2.bas

`Analysis:` Line 4 through 7 define the type that that will be an array within the base type. Lines 11 through 13 define the base type. Darray is dimensioned as a pointer to the type pt since this will be a memory array with the base type. Line 16 defines a pointer to the base type, Dtype, since the base type will also be a memory array. Line 17 defines two working variables, i and j, which will be used in the For-Next loops to create and load data within the types. Line 20 initializes the random number generator.

Line 23 creates a three-element memory using the base type. Lines 25 through 27 then create a three-element memory array using the subtype, pt, for each base type element. Notice that the base type, myType is indexed using the pointer index method since it is a pointer, and each darray element is initialized using Callocate, since darray is also a pointer. Lines 30 through 35 load data into the memory array elements row and column. The outer For loop indexes the base type pointer, while the inner loop indexes the darray type pointer. Since darray is an element of myType, darray is selected using the dot notation, and since row and col are elements of daray, these are selected using the dot notation.

Lines 38 through 43 use the same logic to print the X's on the screen. The outer loop selects the base type pointer, while the inner loop selects the embedded pointer. The row and col elements of darray are used in the Locate statement to move the cursor to the position described by the row and col variables, and an X is printed to the screen.

Lines 46 through 48 deallocate the darray memory array. The order of deallocation is important here. You must deallocate the embedded array first, before you deallocate the base type array. If you deallocate the base type array without deallocating the embedded array, the embedded array elements will remain in memory causing a memory leak. The rule of thumb here is to deallocate the inner most pointer elements first, and then work out toward the base pointer element. Line 50 deallocates the base type elements, only after the darray elements have been deallocated. The program is then closed in the usual way.

When you run the program you should see something similar to the following output. There should be nine X's, however since the random number generator is working within a 10 by 10 space, it is possible two X's may occupy the same row and column, so you may not see all nine X's.

```
   X  X

  X   X

    X

    X
   X  X
X
```

Output 10.7: Output of dyna-type2.bas

While this may seem confusing at first, if you look at the code you will see a recurring pattern. The pointer elements are selected using the pointer index method, followed by the dot notation to select the individual type elements. You start with the base type, in this case myType, index the pointer, and then select the elements using the dot notation, in this case darray. Since darray is a pointer, it is selected using a pointer index, followed by the dot notation to select the row and col elements. The pattern, pointer index, dot notation, pointer index, dot notation would be used for as many levels as needed to resolve a individual element within the memory array.

The following diagram shows the memory layout of the program.

```
MyType[0] -> .darray[0] -> .row, .col
             .darray[1] -> .row, .col
             .darray[2] -> .row, .col
MyType[1] -> .darray[0] -> .row, .col
             .darray[1] -> .row, .col
             .darray[2] -> .row, .col
MyType[2] -> .darray[0] -> .row, .col
             .darray[1] -> .row, .col
             .darray[2] -> .row, .col
```

You can see the pattern reflected in the diagram. If row were a type pointer, then you would just repeat the pattern of index, dot notation on row to resolve the next level of elements. This type of data structure gives you a lot of flexibility in your programming. There is no need to have three darray elements for each base element; myType[0] may point to 4 elements and myType[2] may point to 1 element. You would need to add an additional field in myType to indicate how many darray elements were each myType element, but that is a minor adjustment and easily programmed. What this concept gives you is tight control over your data structures, efficient use of memory and at a relatively low cost in code.

Function Pointers in Types

Once you have created a type definition, you will usually need to create one or more subroutines or functions that act on that data. By using function pointers, you can group the code that acts on this data right along with the data itself. This gives you a powerful capability to organize your data into code objects that operate as a single unit. This is one of the ideas behind OOP, or object oriented programming; encapsulating data and the methods (subroutines and functions) that operate on that data into a single entity. While FreeBasic doesn't yet support object oriented programming, you can derive some of the benefits of OOP by using functions pointers along with data when you create your type definitions.

You define a function (or subroutine) pointer by declaring a prototype function declaration with the type element name. The following code snippet shows a typical declaration.

```
Type myObject
    arg1 As Integer
    arg2 As Integer
    ret As Integer
```

```
        myFunc As Function(arg1 As Integer, arg2 As Integer) As Integer
End Type
Declare Function typeFunc(arg1 As Integer, arg2 As Integer) As Integer
...
Dim Obj as myObject
Obj.myFunc = @typeFunc
...
Function typeFunc(arg1 As Integer, arg2 As Integer) As Integer
        ...
End Function
```

The type definition is defined using data in the normal manner, along with a field, myFunc, that is defined As Function. When a field is defined As Function or As Sub, this creates a pointer to the function or subroutine. Notice that the type definition of myFunc doesn't include a function, but does include the parameter and return types. Since this is a pointer field, the name isn't required, but the parameters and return type in the prototype declaration, must match the actual function in order for the compiler to do type checking on the pointer field.

The Declare statement following the type is needed as a forward reference so that the compiler knows that a function is defined somewhere in the code. If you left out the declaration, you would get a variable not declared error when trying the to compile the code. You create an instance of the type using the Dim statement, just as you have seen in the other examples. Since myFunc is a pointer, you can't use it until you initialize the pointer and you do this by using the Addressof operator on the real function's name. This will store the address of the function in myFunc, which is used when calling the function. The real function must be coded of course, so that you actually have some code to call when using the function pointer.

The following program illustrates creating and using a function pointer in a type def.

```
Option Explicit

'Create a type definition that has data and function ptr
Type myObject
        arg1 As Integer
        arg2 As Integer
        ret As Integer
        myFunc As Function(arg1 As Integer, arg2 As Integer) As Integer
End Type

'Need to declare function for forward reference
Declare Function typeFunc(arg1 As Integer, arg2 As Integer) As Integer
'Create a type variable
Dim Obj As myObject
'Set the address of the function
Obj.myFunc = @typeFunc
```

```
17   'Set the data elements
18   Obj.arg1 = 1
19   Obj.arg2 = 5
20   'Call the function
21   Obj.ret = Obj.myFunc(obj.arg1, Obj.arg2)
22   'Show result
23   Print "Func return is";Obj.ret
24
25   Sleep
26   End
27
28   Function typeFunc(arg1 As Integer, arg2 As Integer) As Integer
29   Return arg1 + arg2
     End Function
```

Listing 10.8: type-func.bas

`Analysis:` Lines 4 through 9 define a type with both integer data fields and a function pointer field. MyFunc is defined As Function along with the function prototype. This sets up myFunc as a function pointer. The Declaration statement in line 12 is used as a forward reference to the actual function code. If you did not supply the declaration, you would get a variable not found error on line 16. Line 14 creates a variable of the type and line 16 uses the Addressof operator to initialize the function pointer to the address of typeFunc, the actual function. Lines 18 and 19 initialize the data fields, arg1 and arg2. Line 21 actually calls the function with the proper arguments. Notice that the dot notation is used in calling the function, myFunc as well as passing the data to the function. Obj.ret olds the functions return value.

The program is closed using the Sleep and End commands, followed by the function definition. Whenever you create a function pointer, you must have some corresponding function code in order to pass the address of that code to the function pointer.

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

```
Func return is 6
```

Output 10.8: Output of type-func.bas

As you can see, function pointers are quite simple to define and use. Not only is your data encapsulated within a single code object, the methods that act on that data are also contained within that same object, giving you a powerful way to organize your data structures. The example program is quite simple of course, but you can use this concept to reduce the complexity of your code. Suppose you are writing a game and the enemy units have been defined using a type definition. By also including the subroutines or functions that act on that data within the type definition, you have a single code object that fully describes an enemy. If you need to make changes to the enemy code, you only have to update a single code object, rather than a scattered bunch of variables and subroutines.

This method also enables you to pass information to functions or subroutines as a single unit, by simply declaring a parameter as the type definition, and you have access to both the data and methods within the called function. This makes the code much more reliable and easier to maintain. It also enables you to generalize the code so that when you create these type of objects, they can be used in other programs.

Forward References

There may be a time when you need to create two type definitions that reference each other. Since FreeBasic is a single pass compiler, this poses a problem since the compiler will encounter a reference to a type that hasn't been defined yet. The solution is to create a forward reference of the second type. You do this by using the Type-As keywords, without the End Type. For example, suppose you have two types Type1 and Type2. Type1 references Type2 and Type2 references Type1. It doesn't matter what order you define the types, you will generate an error in the compiler, because each type has a reference that hasn't been defined yet. In order for the compiler to compile successfully you need to create a forward reference to the second type, and then use that reference in defining the first type. The following code snippet illustrates this concept.

```
'Forward reference
Type FT as Type2

Type Type1
      fType as FT
End Type

Type Type2
      fType as Type2
End Type
```

The code `Type FT as Type2` creates the forward reference that is in turn used in the Type1 definition to refer to Type2, `fType as FT`. FT and Type2 are actually the same thing, FT is just an alias for the Type2 definition. Whenever you need to have one or mote type definitions refer to each other, you will need to create forward declarations for the the types that have not been defined when referenced.

Bit Fields

There is yet another data type that can be used in type definitions, the bit field. Bit fields are defined as *variable_name*: *bits* As DataType. The variable name must be followed with a colon and the number of bits, followed by the data type. Only integer data types are allowed within a bit field. Bit fields are useful when you need to keep track of boolean type information, such as if a pixel is on or off. The following program illustrates using bit fields within a type definition.

```
Option Explicit

```

```
3    Type BitType
4        b1: 1 As Integer
5        b2: 4 As Integer
6    End Type
7
8    Dim myBitType As BitType
9
10   myBitType.b1 = 1
11   myBitType.b2 = 1101
12
13   Print "Bit field 1: ";myBitType.b1
14   Print "Bit field 2: ";myBitType.b2
15
16   Sleep
17   End
```

Listing 10.9: bitfield.bas

Analysis: Lines 3 through 6 define a type with two bit fields. B1 is defined as 1 bit, and b2 is defined as 4 bits. Line 8 creates a variable of the type definition. Line 10 sets b1 to 1. Since b1 is defined a 1 bit, the only valid values are 0 or 1. Line 11 sets b2 to 1101. Here there are four bits so you can have a range of 0000 to 1111. Lines 13 and 14 print out the values of the bits. The program is closed in the usual way.

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

```
Bit field 1: 1
Bit field 2: 13
```

Output 10.9: Output of bitfield.bas

The data type of the bit field determines how many bits you can declare in a bit field. Since an integer is 32 bits long, you could declare up to 32 bits in the field. However, in most cases you would declare a single bit for each field, and use a number of fields to define the bit masking that you wish to use. Using a single bit simplifies the coding you need to do to determine if a bit is set or cleared.

The Field Property

When you create a variable of a type definition, the type is padded in memory. The padding allows for faster access of the type members since the type fields are aligned on a 4 byte or Word boundary. However, this can cause problems when trying to read a type record from a file that is not padded. You can use the use field property to change the padding of a type definition. The field keyword is used right after the type name and can have the values 1, for 1 byte alignment (no padding), 2 for 2 byte alignment and 4 for 4 byte alignment. To define a type with no padding you would use the following syntax.

```
Type myType field = 1
      v1 As Integer
      v2 As Byte
End Type
```

For 2 byte alignment you would use field = 2. If no field = property is assigned, then the padding will be 4 bytes. If you are reading a type definition created by FreeBasic using the default alignment, then you do not need to use the field property.

If you reading a Quick Basic type record, then you will need to use field = 1, as QB used byte alignment by default.

Type Initialization

You can initialize a type definition when you dimension the type just as you can any of the intrinsic variables. The following code snippet illustrates the syntax.

```
Type aType
      a As Integer
      b As Byte
      c As String
End Type
Dim myType As aType => (12345, 12, "Hello")
```

In the Dim statement, the arrow operator is used to signal the compiler that you are initializing the type variable. The type element values must be enclosed in parenthesis, and separated by commas. The order of the value list corresponds to the order of the type elements, where a will be set to 12345, b to 12 and c to “Hello”. The following short program initializes a type using this syntax.

```
Option Explicit

'Create a type def
Type aType
    a As Integer
    b As Byte
    c As String
End Type

'Create and init the type
Dim myType As aType => (12345, 12, "Hello")

'Display values
With myType
```

```
15       Print .a
16       Print .b
17       Print .c
18   End With
19
20   Sleep
21   End
```

Listing 10.10: type-init.bas

Analysis: Lines 4 through 8 define a type definition. Line 11 dimensions the type variable and sets the filed elements to the listed values. The order of the values correspond to the order of the field elements. Line 14 through 18 prints the type values to the screen. The program is then closed in the usual way.

When you run the program you should the following output.

```
 12345
 12
Hello
```

Output 10.10: Output of type-init.bas

Initializing a type definition in a Dim statement is useful when you need to have a set of initial values for a type, or values that will not change during program execution. Since the values are known at compile time, the compiler can doesn't have to spend cycles loading the values during runtime.

Unions

Unions look similar to Types in their definition.

```
Union aUnion
      b As Byte
      s As Short
      i As Integer
End Union
```

If this were a Type, you could access each field within the definition. For a Union, you can only access one field at any given time; all the fields within a Union occupy the same memory segment, and the size of the Union is the size of the largest member. In this case, the Union would occupy four bytes, the size of an Integer, with the b field occupying 1 byte and the s field occupying 2 bytes within the 4 byte integer space. Each field starts at the first byte, so the s field would include the b field, and the i field would include both the b field and the s field. The following program illustrates this concept.

```
1    Option Explicit
2
3    'Define a union
4    Union aUnion
5        b As Byte
6        s As Short
7        i As Integer
8    End Union
9
10   Dim myUnion As aUnion
11
12   'Set the integer value
13   myUnion.i = 2047483641
14   'Print members
15   Print "Integer: ";Tab(10);Bin(myUnion.i)
16   Print "Byte: ";Tab(33);Bin(myUnion.b)
17   Print "Short: ";Tab(28);Bin(myUnion.s)
18
19   Sleep
20   End
```

Listing 10.11: union.bas

Analysis: Line 4 through 8 define the example union. Line 10 defines a variable of the union definition. Line 13 sets the integer field to a value that will overlap the b and s fields. Lines 15 through 17 print out the values of each field of the union in binary so that you can see the overlap of the values. The program is closed in the usual way.

When you run the program you should the following output.

```
Integer: 1111010000010100001111011111001
Byte:                           11111001
Short:                     1111011111001
```

Output 10.11: Output of union.bas

You can easily see the overlapped values in the output. While a union is useful on its own, you can combine a union with a type definition to create extremely flexible and efficient data structures.

Types in Unions

A good example of using a type definition in a union is the Large_Integer definition found in winnt.bi. The Large_Integer data type is used in a number of Windows functions within the C Runtime Library. The following code snippet shows the Large_Integer definition.

```
union LARGE_INTEGER
	type
		LowPart as DWORD
		HighPart as LONG
	end type
	QuadPart as LONGLONG
end union
```

The Dword data type is defined in windef.bi as an Uinteger, `type DWORD as uinteger`, and the Longlong type is defined as a Longint, `type LONGLONG as longint`. A Long is the same as an integer. Remember that a type occupies contiguous memory locations, so the HighPart field follows the LoPwart part field in memory, the type occupies the same memory segment as the QuadPart field. When you set QuardPart to a large integer value, you are also setting the values of the type fields, which you can then extract as the LowPart and HighPart. You can also do the reverse, that is by setting the LowPart and HighPart of the type, you are setting the value of the QuadPart field.

As you can see, using a type within a union is an easy way to set or retrieve individual values of a component data type without resorting to a lot of conversion code. The layout of the memory segments does the conversion for you, providing that the memory segments make sense within the context of the component type. In the Large_Integer case, the LowPart and HighPart have been defined to return the appropriate component values. Using values other than Dword and Long would not return correct values for LowPart and HighPart. You need to make sure when defining a type within a union, you are segmenting the union memory segment correctly within the type definition.

Unions in Types

A union within a type definition is an efficient way to manage data when one field within a type can only be a single value. The most common example of this is the Variant data type found in other programing languages.

FreeBasic does not have a Variant data type at this time. However, when classes are added to FreeBasic, it will be quite easy to create a Variant data type and overload the arithmetic operators to create a Variant that behaves just like the intrinsic data types. Classes are planned for a future version of FreeBasic.

If you can only access a single field within a union, how do you know which field to access when a union is contained within a type definition? Usually you will include an identifier field within the type, but outside the union, that identifies which union field to access. The following program illustrates this concept by creating a simple Variant data type.

```
Option Explicit

'Union field ids
#define vString 0
```

```
5   #define vInteger 1
6
7   'Define type def with variable data fields
8   Type vType
9       vt_id As Integer
10      Union
11          s As String
12          i As Integer
13      End Union
14  End Type
15
16  'Create variant variable
17  Dim myVariant As vType
18
19  'This subroutine prints out value
20  Sub PrintVariant(v As vType)
21      If v.vt_id = vString Then
22          Print "String value: ";v.s
23      Elseif v.vt_id = vInteger Then
24          Print  "Integer value:";v.i
25      End If
26  End Sub
27
28  'Set the id to a string
29  myVariant.vt_id = vString
30  myVariant.s = "This is a string."
31  'Print the string value
32  PrintVariant myVariant
33  'Clear the string memory before seting the integer
34  myVariant.s = ""
35  'Set the id to an integer
36  myVariant.vt_id = vInteger
37  myVariant.i = 300
38  'Print the integer value
39  PrintVariant myVariant
40
41  Sleep
42  End
```

Listing 10.12: simplevariant.bas

`Analysis:` Line 4 and 5 define the union ids. These values will be used to determine which field within the union to access. Lines 20 through 26 define a subroutine that will print the value of the union field based on the id. If the id field is VString, then the s field within the union is printed. If the id field is VInteger, then the i field is printed. Line 29 sets the id field to Vstring and line 30 sets the union field s to a string. The PrintVariant subroutine is called in line 32 to print out the value. Line 34 clears the allocated memory of the dynamic string by setting the union field s to an empty string.

Line 36 sets the id field to an integer, and the value of the union field i is set to 300. Line 39 calls the PrintVariant subroutine to print the integer value. The program is then closed in the usual way.

Caution There are two subtle danger in this program that may not be readily apparent. Remember that a dynamic string is actually a pointer to an allocated memory segment. The union field s is actually a pointer to this memory segment. Since the fields s and i overlap in memory, setting i to a value also sets s to a value; that is, in the program i is set to 300, which also sets s to 300. Since s is a pointer, trying to access s after setting i will access memory location 300, which will be garbage data and may cause the program to crash.

The other problem with this program is that it can potentially lead to a memory leak while the program is running. The string value is an allocated memory segment. Setting the integer value, overwrites the pointer address in s, which means you have lost the pointer address to the memory segment. The string data still exists in memory, but now that data is lost, causing a memory leak. This is why the string value in the union was set to an empty string before setting the integer value. Setting the string value to an empty string deallocates the string data and frees the string pointer.

When using pointers within a union, great care must be taken to insure that the pointer is pointing to the correct memory location, and that they are deallocated correctly to prevent memory leaks.

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

```
String value: This is a string.
Integer value: 300
```

Output 10.12: Output of simplevariant.bas

Using a combination of unions and types within a program allows to design custom data types that have a lot of flexibility, but care must be taken to ensure that you are using the data constructs correctly. Improper use of these data types can lead to hard-to-find bugs. The benefits however, out-weigh the risks and once mastered, are a powerful programming skill.

A Look Ahead

There are times when you will need values that do not change during a program. These values are called Symbolic Constants and are the subject of the next chapter.

11 Symbolic Constants

Constants are values that do not change during the life of the program. Constants are like variables in that the name of the constant refers to the defined value, but unlike a variable, these values cannot be changed in the program. There are two important reasons why constants should be used in your program. One, they help to document the program. Suppose in your role-playing game you have a weapon such as a broadsword. If you define the broadsword as a constant, which you can do with `#Define broadsword 12`, you can then refer to the weapon id as `broadsword`, rather than 12. The number 12 imparts no real information when you see it in the code; `broadsword` on the the other hand is quite clear and understandable.

The second reason to use a constant is code maintenance. There may come a time when working on your role-playing game that you need to change the value of the broadsword id. If you have defined the id as a constant, you only need to change it in a single location, the place where you defined the constant. If you had just used the number 12, you would have to search through the code and change each instance where 12 referred to the broadsword. If the program is of any length at all, you will probably miss a reference or two, introducing bugs into your program. Bugs that may be difficult to locate and fix.

One of the things you will discover as you progress in your programming adventure is that programs are dynamic, not static. There is always a new technique being developed that you can use, a new compiler function that will improve your program, and bugs that need to be fixed. The only thing that stays the same in a program is the fact that programs continually evolve over time. You should always keep in mind that when you write a program, you will probably end up making changes to the program, and you should code accordingly. It may be easier to write 12 than it is to write broadsword, but a few extra seconds of typing will save you hours when you need to change 12 to 120.

#Define as a Constant

You have already seen #Define at work in several of the example programs. #Define is a preprocessor command, where the defined symbol is physically replaced in the code by the associated value. #Define of course is used in a number of situations, from creating macros to conditional compilation, but it is also useful for creating constants within your program. If you look through the declaration files in the include folder of your FreeBasic installation, you will see that #Define is used extensively for constant values. Since the compiler replaces the symbol definition with the defined value, it is a very efficient coding method. It is also quite easy to use, as the following code snippet illustrates.

```
'Define directions
#Define north 1
#Define neast 2
#Define east 3
#Define seast 4
```

```
#Define south 5
#Define west 6
#Define swest 7
#Define nwest 8
```

Once you define the constants, you can use the symbols in your programs just as you would the values. You have already seen this technique used in some of the example programs presented in the book.

The Const Keyword

The Const keyword is another method to define constants in your program. The format is similar to the #define, as the following code snippet illustrates.

```
Const xk = Chr(255)
Const key_up = xk & Chr(72)
Const key_dn = xk & Chr(80)
Const key_rt = xk & Chr(77)
Const key_lt = xk & Chr(75)
```

These constants are the character codes returned by Inkey for the arrow keys. Inkey returns a two byte string for the extended keys, Chr(255) + the extended key character code. Inkey is covered in detail in the chapter on Console Programming. To use these constaints in your program you would just use key_lt for example to check to see if the left arrow key had been pressed.

Const Versus #Define

As you can see Const and #Define are similar constructs. The question then becomes, which one should you use? Remember that #Define replaces the symbol name with the text following the symbol. If you wrote `#Define key_up xk & Chr(72)`, then the code `xk & Chr(72)` would be replace the symbol `key_up`. If you had several places where you used `key_up`, then there would be several instances of `xk & Chr(72)` in your program. You can see that by using `#Define` in this case your program would be performing the same calculation over and over. It is much more efficient to use Const in this case, since the calculation is only done once, when the Const is defined.

On the other hand, if you are defining single constant values, such as compass directions, then using a #Define is preferable to using Const. For a Const value, the compiler must do a lookup in the symbol table and it is much more efficient to simple replace the symbol with the value using #Define.

Enumerations

Enumerations are sequential values that the compiler can calculate for you. To create an enumeration, you use enclose the values within an Enum-End Enum block. The compass direction defined above could also be defined as an enumeration.

```
Enum compass
       north = 1
```

```
        neast
        east
        seast
        south
        west
        swest
        nwest
End Enum
```

In this example, neast will be defined as 2, with east defined as 3 and so on. If no starting value is set, enumerations start at 0. You can also change the sequence within an enumeration by setting a symbol to a value using =, and any following symbols will be incremented from this starting point.

```
Enum compass
        north = 1
        east
        south
        west
        neast = 10
        seast
        swest
        nwest
End Enum
```

In this example, the value of seast will be 11, swest will be 12 and so on. Once you define an enumeration, you can create variables of the Enum and use that variable within your program.

```
Dim aCompass as compass
```

You can then use the enumeration values to initialize the variable. The following code snippet set aCompass to the north-defined value.

```
aCompass = north
```

Caution The compiler does not check to see if the value being passed to the Enum variable is within the defined range of the enumeration. It is the responsibility of the programmer to ensure that the Enum variable contains the correct values.

A Look Ahead

In the next chapter you will see how to create and work with Arrays in FreeBasic.

12 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 10 columns, 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.

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

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

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

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

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, if Option Base has not been defined. If Option Base 1 had been used, then the row indexes would range from 1 to 2, and the column indexes would range from 1 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.

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

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

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

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

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.

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

```
1    Option Explicit
2
3    'Create a two-dimensional array
4    Dim As Integer myArray(1 To 2, 1 To 10), i, j
5
6    'Load some data into the array
7    For i = 1 To 2
8        For j = 1 To 9
9            myArray(i, j) = Rnd * 10
10       Next
11   Next
12
13   'Print data in array
14   For i = 1 To 2
15       For j = 1 To 9
16           Print "row:";i;" col:";j;" value:";myArray(i, j)
17       Next
18   Next
19
20   Sleep
21   End
```

Listing 12.1: arrayindex.bas

`Analysis:` Line 4 creates a two-dimensional array with two rows, with each row having ten columns. The working variables i and j are also declared on the same line. Lines 7 through 11 load some random data into the array. A nested For-Next loop 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.

Lines 14 through 18 will 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: 6
```

```
row: 1 col: 3 value: 2
row: 1 col: 4 value: 8
row: 1 col: 5 value: 6
row: 1 col: 6 value: 5
row: 1 col: 7 value: 4
row: 1 col: 8 value: 9
row: 1 col: 9 value: 8
row: 2 col: 1 value: 7
row: 2 col: 2 value: 2
row: 2 col: 3 value: 9
row: 2 col: 4 value: 7
row: 2 col: 5 value: 5
row: 2 col: 6 value: 3
row: 2 col: 7 value: 0
row: 2 col: 8 value: 1
row: 2 col: 9 value: 4
```

Output 12.1: Output of arrayindex.bas

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.

In order to use dynamic arrays in your program you need to include the `Option Dynamic` directive in your program. Static arrays, the arrays described above, are kept on the heap, but dynamic arrays are allocated from the computer's pool of memory so Option Dynamic is needed to tell the compiler to dynamically allocate the array. 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 th 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 without specifying Option Dynamic.

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

```
Option Explicit
Option Dynamic
```

```

'Create an integer stack. The 0 index will be our empty marker.
Dim As Integer stack(0), top = 0, i, ivalue

'This will push a value on the stack, update the top of stack
Sub PushValue(astack() As Integer, stacktop As Integer, value As Integer)
    'Increment the top of the stack
    stacktop += 1
    'Resize the stack
    Redim Preserve astack(stacktop)
    astack(stacktop) = value
End Sub
'This will pop a value off the stack, update top of stack
Function PopValue(astack() As Integer, stacktop As Integer) As Integer
    Dim As Integer ret

    If stacktop = 0 Then
        ret = 0
    Else
        ret = astack(stacktop)
        stacktop -= 1
        Redim Preserve astack(stacktop)
    End If
    Return ret
End Function

'Push five values on to stack
Print "Pushing values on stack..."
For i = 1 To 5
    ivalue = i * 2
    PushValue stack(), top, ivalue
    Print "Stack top:";Ubound(stack),"Value:";ivalue
Next
Print

'Pop the values off the stack
Print "Popping values from stack..."
ivalue = PopValue(stack(), top)
Do While ivalue > 0
    Print "Stack Value:";ivalue
    ivalue = PopValue(stack(), top)
Loop
Print

'Check stack size
```

```
48   Print "Stack size after pops:";Ubound(stack)
49
50   Sleep
51   End
```

Listing 12.2: stack.bas

Analysis: 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.

Line 10 increments the stack top variable, and line 12 uses Redim Preserve to resize the array, while keeping the existing data in the array. Line 13 set the new array location to the value passed to the subroutine.

Line 16 through 27 pops the top value of the stack, if the stack index is grater than zero, updates the top-of-stack variable and then returns the popped value. Line 19 and 20 checks to make sure that the stack has some data. If the index is 0, the stack is empty. Lines 22 through 24 get the current value of the top stack item, updates the stack pointer variable and then resizes the stack using Redim Preserve. Lines 31 through 35 use a For-Next loop to push 5 values on to the stack. Lines 40 through 44 pop the values off the stack. A Do-Loop is used to pop the values since when the stack is empty, the return value will be 0. Line 48 then cheks to see if all the values have been popped from the stack.

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 Value: 10
Stack Value: 8
Stack Value: 6
Stack Value: 4
Stack Value: 2

Stack size after pops: 0
```
Output 12.2: Output of 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

Function	Syntax	Comment
		1, the second dimension is 2, and so on.
Option Dynamic	Option Dynamic	Allocates arrays into dynamic memory by default.
Option Static	Option Static	Clears a previous Option Dynamic. All subsequent arrays will be allocated on the heap.
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 12.1: 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 the size of a two dimensional array.

```
Option Explicit
Option Dynamic

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

```
16  Print
17  Print "Additional Redims"
18  Print "-----------------"
19  'Redim array five times
20  For i = 1 To 5
21      nb = Ubound(myArray, 1) + 1
22      Redim myArray(1 To nb, 1 To 5)
23      'Print new array size
24      Print "Min Row Index:";Lbound(myArray, 1);
25      Print " -- Max Row Index:";Ubound(myArray, 1)
26      Print "Min Column index:";Lbound(myArray, 2);
27      Print " -- Max Column index:";Ubound(myArray, 2)
28      Print
29  Next
30
31  Sleep
32  End
```

Listing 12.3: ulbound.bas

Analysis: Line 2 set the compiler directive Option Dynamic so that the array is allocated in dynamic memory. Line 5 creates a dynamic array with no dimensions specified. Line 7 uses Redim to create a two-dimensional array. Lines 10 through 15 print the upper and lower bounds 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.

Lines 20 through 29 resize the first dimension of the array and print the new lower and upper indexes. Line 21 gets the current upper bound of the first dimension, 1 is added to that value and then Redim is used in line 22 to resize the first dimension of the array. Lines 24 through 27 get the new bounds and print the values 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
```

Listing 12.4: Output of 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.

Arrays of Types

Type definitions allow you to group related data into a single entity, and often you will need more than one instance of a type to fully express the data. Arrays of types allow you create multiple instances of a type definition that can be easily managed using the arrays functions. An example of this usage may be an inventory system for your RPG, a series of document descriptions within an editor, and a set of employee records from a random access database.

You create arrays of types just as you would with any of the intrinsic data types. The following code snippet illustrates the syntax.

```
Type myPoint
       x As Integer
       y As Integer
End Type
Type myLine
       p1 As myPoint
       p2 As myPoint
       char As String * 1
End Type
Dim myLineSet (1 to 3) as myLine
```

The code defines a set of 3 lines, with endpoints p1 and p2, where each endpoint is located at row and col. The following program illustrates using this definition to print some lines to the console screen.

```
Option Explicit

'Define the point
Type myPoint
       x As Integer
       y As Integer
End Type

'Define the line
Type myLine
       p1 As myPoint
       p2 As myPoint
       char As String * 1
End Type

'Create a set of 3 lines
Dim myLineSet (1 To 3) As myLine
Dim As Integer i

'This subroutine uses the Bresenham Line algorithm
'to print a line on the console screen. Google
'"Bresenham Line algorithm" for more information.
Sub DrawLine(aLine As myLine)
    Dim As Integer i, deltax, deltay, num
    Dim As Integer d, dinc1, dinc2
    Dim As Integer x, xinc1, xinc2
    Dim As Integer y, yinc1, yinc2
    Dim As Integer x1, y1, x2, y2

    'Get the endpoint coordinates
    x1 = aLine.p1.x
    y1 = aLine.p1.y
    x2 = aLine.p2.x
    y2 = aLine.p2.y

    'Get the delta change in both x and y
    deltax = Abs(x2 - x1)
    deltay = Abs(y2 - y1)

    'Calculate the slope of the line
    If deltax >= deltay Then
```

```
            num = deltax + 1
            d = (2 * deltay) - deltax
            dinc1 = deltay Shl 1
            dinc2 = (deltay - deltax) Shl 1
            xinc1 = 1
            xinc2 = 1
            yinc1 = 0
            yinc2 = 1
        Else
            num = deltay + 1
            d = (2 * deltax) - deltay
            dinc1 = deltax Shl 1
            dinc2 = (deltax - deltay) Shl 1
            xinc1 = 0
            xinc2 = 1
            yinc1 = 1
            yinc2 = 1
        End If

        If x1 > x2 Then
            xinc1 = - xinc1
            xinc2 = - xinc2
        End If

        If y1 > y2 Then
            yinc1 = - yinc1
            yinc2 = - yinc2
        End If

       x = x1
       y = y1
      Locate y, x
      Print aLine.char;
        For i = 2 To num
          'Get the next iteration of the line
          If d < 0 Then
              d = d + dinc1
              x = x + xinc1
              y = y + yinc1
          Else
              d = d + dinc2
              x = x + xinc2
              y = y + yinc2
            End If
          Locate y, x
```

```
87         Print aLine.char;
88       Next
89   End Sub
90
91   'Returns a random number between low and high
92   Function GetRandom(lowerbound As Integer, upperbound As Integer) As Integer
93      GetRandom = Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
94   End Function
95
96   Randomize Timer
97   'Initialize and print the lines
98   For i = Lbound(myLineSet) To Ubound(myLineSet)
99       'Set the first endpoint coordinates
100      myLineSet(i).p1.x = GetRandom(1, 80)
101      myLineSet(i).p1.y = GetRandom(1, 25)
102      'Set the second endpoint coordinates
103      myLineSet(i).p2.x = GetRandom(1, 80)
104      myLineSet(i).p2.y = GetRandom(1, 25)
105      'Get the display character
106      myLineSet(i).char = Chr(GetRandom(33, 47))
107      'Print the line
108      DrawLine myLineSet(i)
109  Next
110
111  Sleep
112  End
```

Listing 12.5: arrayoftypes.bas

`Analysis:` Lines 4 through 7 define the coordinates for the an endpoint of a line. Lines 10 through 14, define a line, with two endpoints and the character used to print the line on the console screen. Line 17 defines a set of three lines, and line 18 defines a working variable that will be used in a For-Next loop that will initialize and display the line.

Lines 23 through 89 define a subroutine that implements the Bresenham Line algorithm to print the line to the console screen. There are a number of resources on the Internet that explain the algorithm in detail and can be found using your favorite search engine.

Lines 92 through 94 define a function that will return a random integer in the range of lowerbound to upperbound, inclusive. Line 96 initializes the random number generator by seeding the generator with the current Timer value. Randomize Timer ensures that each call to Rnd will return a different random number. Without the Randomize statement, Rnd will return the same sequence of random number.

Line 98 through 109 initialize each line in the array and then calls the DrawLine subroutine to print the line. When accessing the fields of a type within an array, you must specify which array element you are accessing by using parentheses with an index value, jut as you would an intrinsic data type. You can then access each element within the type using the familiar dot notation you saw in the chapter on composite types. Notice that the syntax is similar to the syntax used in the type memory array example, and the rules are accessing type fields is the same.

Once each array element has been initialized and each line printed, the program is closed in the usual way.

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

```
                        %%%
                     %%%
                   %%                                )
                %%%                                  )
             %%%                                     )
          %%%                                        )
       %%%                                           )
     %%                                              )
  %%%                                                )
%%                                                   )

             //////
                   //////////
                             //////////
                                       //////////
                                                 //////////
                                                           //////////
                                                                     /////
```

Output 12.3: Output of arrayoftypes.bas

Your output will vary since the end points of the lines, and the line character are random generated.

Arrays in Types

As you saw in the chapter on composite types, type fields be both intrinsic and composite type. You can also create static arrays within a type definition. In the listing above, you can see that a line has two endpoints. Rather than using two distinct fields, you can create an array of endpoints that describe the line. The following code listing is a modification of the previous listing, using an array to describe the endpoints.

```
Option Explicit

```

```
'Define the point
Type myPoint
       x As Integer
       y As Integer
End Type

'Define the line
Type myLine
    pts(1 To 2) As myPoint
    char As String * 1
End Type

'Create a set of 3 lines
Dim myLineSet (1 To 3) As myLine
Dim As Integer i, j

'This subroutine uses the Bresenham Line algorithm
'to print a line on the console screen. Google
'"Bresenham Line algorithm" for more information.
Sub DrawLine(aLine As myLine)
    Dim As Integer i, deltax, deltay, num
    Dim As Integer d, dinc1, dinc2
    Dim As Integer x, xinc1, xinc2
    Dim As Integer y, yinc1, yinc2
    Dim As Integer x1, y1, x2, y2

    'Get the endpoint coordinates
    x1 = aLine.pts(1).x
    y1 = aLine.pts(1).y
    x2 = aLine.pts(2).x
    y2 = aLine.pts(2).y

    'Get the delats change in both x and y
    deltax = Abs(x2 - x1)
    deltay = Abs(y2 - y1)

    'Calculate the slope of the line
    If deltax >= deltay Then
        num = deltax + 1
        d = (2 * deltay) - deltax
        dinc1 = deltay Shl 1
        dinc2 = (deltay - deltax) Shl 1
        xinc1 = 1
        xinc2 = 1
        yinc1 = 0
```

```
            yinc2 = 1
        Else
            num = deltay + 1
            d = (2 * deltax) - deltay
            dinc1 = deltax Shl 1
            dinc2 = (deltax - deltay) Shl 1
            xinc1 = 0
            xinc2 = 1
            yinc1 = 1
            yinc2 = 1
        End If

        If x1 > x2 Then
            xinc1 = - xinc1
            xinc2 = - xinc2
        End If

        If y1 > y2 Then
            yinc1 = - yinc1
            yinc2 = - yinc2
        End If

        x = x1
        y = y1
      Locate y, x
      Print aLine.char;
        For i = 2 To num
          'Get the next iteration of the line
          If d < 0 Then
              d = d + dinc1
              x = x + xinc1
              y = y + yinc1
          Else
              d = d + dinc2
              x = x + xinc2
              y = y + yinc2
            End If
          Locate y, x
          Print aLine.char;
        Next
    End Sub

    'Returns a random number between low and high
    Function GetRandom(lowerbound As Integer, upperbound As Integer) As Integer
       GetRandom = Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
```

```
93   End Function
94
95   Randomize Timer
96   'Initialize and print the lines
97   For i = Lbound(myLineSet) To Ubound(myLineSet)
98       For j = Lbound(myLineSet(i).pts) To Ubound(myLineSet(i).pts)
99           'Set the first endpoint coordinates
100          myLineSet(i).pts(j).x = GetRandom(1, 80)
101          myLineSet(i).pts(j).y = GetRandom(1, 25)
102          'Set the second endpoint coordinates
103          myLineSet(i).pts(j).x = GetRandom(1, 80)
104          myLineSet(i).pts(j).y = GetRandom(1, 25)
105          'Get the display character
106          myLineSet(i).char = Chr(GetRandom(33, 47))
107          'Print the line
108      Next
109      DrawLine myLineSet(i)
110  Next
111
112  Sleep
     End
```

Listing 12.6: arrayintypes.bas

`Analysis:` This program is identical to the previous program, except that this program uses an array to describe the endpoints of the line. Only the changes to the program will discussed in this section.

Line 11 defines a single dimension array of endpoints that has two elements. In lines 97 through 110, the program initializes each line element. The outer loop selects each line type within the line set array. The inner For-Next loop then selects each endpoint element with the endpoint array. Notice that you an use Lbound and Ubound for this array, just as you can with the main array. Arrays within a type definition are true arrays, and all the functions available in FreeBasic can be used with embedded arrays. The inner j loop Lbound and Ubound use the outer loop specification, `myLineSet(i).pts`, to select a line element within the line set array, and then the function is applied to the embedded array. The endpoints are then initialized using both the i and j specifications, `myLineSet(i).pts(j).x = GetRandom(1, 80)`. The variable i selects a line while j selects an endpoint within the selected line. The dot notation is used to reference each field within the type definition, including the endpoint array. Again, the notation and principles are similar to a type memory array.

The program is closed in the usual way after printing all the lines.

The output of this program is the same as the previous program and is not shown here. Using an array for the endpoints enables you to easily extend the line definition to support not only lines, but triangles and squares. The following code snippet shows one possible definition.

```
Type myObj
	objid As Integer
	Union
		myLine(1 To 2) As myPoint
		myTriangle(1 To 3) As myPoint
		mySquare(1 To 4) As myPoint
	End Union
	char As String * 1
End Type
```

The objid field would indicate which type of object is contained within the definition. That is, a 1 may indicate a line, a 2 may indicate a triangle and a 3 may indicate a square. Since the definition defines a single object, a union is used to enclose the endpoint arrays to maximize memory usage. To print the object to the screen, you would examine the objid and then use the Lbound and Ubound on the appropriate endpoint array definition, printing the number of lines that correspond to the type of object. One further enhancement you can make to this program is to add a function pointer to the type definition, and then write print routines that correspond to the type of object being printed. Using this technique will enable you to further extend the usefulness of the code by simplifying the process of adding new objects to the type definition. For example, if you needed to be able to describe a cube, you would simply add an new array to the union, add a cube print function, and the type definition would be able to print a cube by simply adding a few lines of code, while keeping the original functionality intact.

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.

```
1     Option Explicit
2
3     Dim aArray(1 To 5) As Integer => {1, 2, 3, 4, 5}
4     Dim bArray(1 To 2, 1 To 5) As Integer => {{1, 2, 3, 4, 5}, _
5                                               {6, 7, 8, 9, 10}}
6     Dim As Integer i, j
7
8     'Print aArray values.
9     For i = 1 To 5
10        Print "aArray(";i;" ):";aArray(i)
11    Next
12    Print
13
14    'Print bArray values.
15    For i = 1 To 2
16        For j = 1 To 5
17            Print "bArray(";i;",";j;" ):";bArray(i, j)
18        Next
19    Next
20
21    Sleep
22    End
```

Listing 12.7: arrayinit.bas

Analysis: Line 3 creates a single-dimension array and initializes the values of the array. Line 4 creates a two-dimensional array and initializes the both rows of the array. Line 9 through 11 print the contents of the first array to the screen and lines 15 through 19 print the contents of the second to the screen. The program is then closed in the usual way.

When you run the program 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
```

Listing 12.8: Output of 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.

Type Array Initialization

Since you can initialize a type definition and an array, it stands to reason you can do the same with a type array. In the chapter on type definitions you saw the syntax for initializing a type, and here you have seen how to initialize an array. To initialize a type array, you use both the type initialization syntax with the array initialization syntax, as you would expect. The following code snippet illustrates this combined syntax.

```
Type aType
       a As Integer
       b As Byte
       c(1 To 2) As Zstring * 10
End Type
Dim As aType myType(1 To 2) => { (1234, 12, {"Hello", "World"}), _
                                  (5678, 24, {"From", "Freebasic"}) _
                                }
```

The curly brackets signify that this is an array initialization, while the parenthesis indicate the type initialization. Since the type has an embedded array, you use the curly brackets to load the data into the embedded array, just as you would a stand-alone array. If the embedded array was a multidimensional array, then you would need to wrap each row in { and } just as you would a stand-alone array.

The following program shows how all this works together.

```
Option Explicit

'Create a type definiton
Type aType
    a As Integer
    b As Byte
    c(1 To 2) As Zstring * 10
```

```
8    End Type
9
10   'Dimension and init the type array
11   Dim As aType myType(1 To 2) => { (1234, 12, {"Hello", "World"}), _
12                                    (5678, 24, {"From", "Freebasic"}) _
13                                  }
14   'Working variables
15   Dim As Integer i, j
16
17   'Display type values
18   For i = 1 To 2
19       Print "Type array index: ";i
20       With myType(i)
21           Print .a
22           Print .b
23           For j = 1 To 2
24               Print .c(j)
25           Next
26       End With
27       Print
28   Next
29
30   Sleep
31   End
```

Listing 12.9: typearray-init.bas

`Analysis:` Lines 4 through 8 define the type definition. Line 11 dimensions and initializes the type definition array. The type elements are enclosed in parenthesis, while the embedded array is enclosed within curly brackets. Line 15 creates some working variables for the For-Next loops. Lines 18 through 28 print the values to the console screen using a With block. The program is closed in the usual way.

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

```
Type array index:  1
 1234
 12
Hello
World

Type array index:  2
 5678
 24
From
Freebasic
```

Output 12.4: Output of typearray-init.bas

CRT Array Functions

There are a few CRT functions that you can use to manipulate arrays. The following table lists the Mem* functions which are defined in string.bi.

Function	Syntax	Comment
Memchr	Ptr = Memchr(@Array(start), byte_value, Ubound(Array))	Returns a pointer to byte_value in byte Array, or Null pointer if byte_value is not found. Memchr only works with byte arrays.
Memcpy	Ptr = Memcpy(@ArrayTo(start), @ArrayFrom(start), num_bytes)	Copies num_bytes from ArrayFrom to ArrayTo and returns a pointer to ArratTo. Do not use this function if From and To overlap. Use Memmove instead.
Memmove	Ptr = Memmem(@ArrayTo(start), @ArrayFrom(start), num_bytes)	Works the same as Memcpy but correctly handles overlapping memory segments.

Table 12.2: CRT Array Functions

The following program illustrates using Memchr to search for a byte within a byte array, and Memcpy to copy one array into another array.

```
Option Explicit

#include once "crt.bi"

```

```
5    Dim As Ubyte aArray(1 To 5) => {65, 123, 100, 52, 42}, bArray(1 To 5)
6    Dim bret As Ubyte Ptr
7    Dim As Integer index, i
8
9    'Look for 100 in byte array
10   Print "Looking for byte 100..."
11   Print
12   bret = Memchr(@aArray(1), 100, Ubound(aArray))
13   If bret <> NULL Then
14       index = bret - @aArray(Lbound(aArray)) + Lbound(aArray)
15       Print "100 found at index ";index
16   Else
17       Print "100 not found."
18   End If
19   Print
20   'Copy aArray to bArray
21   Print "Copying aArray to bArray..."
22   Print
23   bret = Memcpy(@bArray(1), @aArray(1), Ubound(aArray) * Sizeof(Ubyte))
24   If bret <> NULL Then
25       For i = 1 To 5
26           Print "bArray(";i;" ) = ";bArray(i)
27       Next
28   End If
29
30   Sleep
31   End
```

Listing 12.10: memfunc.bas

`Analysis:` Line 3 includes the crt.bi header file so that Memchr and Memcpy will be available to the program. Both functions are located in string.bi, which is included when you include crt.bi. Line 5 creates two ubyte arrays, the first which is initialized with some byte data. Remember that the ubyte data type has a range of 0 to 255. Line 6 declares a ubyte pointer which will be the return value fr the Memchr function. Line 7 declares two working variables, index which will be used to calulate the array index of the Memchr search, and I which will be used to print the contents of bArray after the Memcpy operation.

Line 12 calls the Memchr function. The first parameter is the address of the first element of aArray, which has an index of 1, the value to find, which is 100, and the length of the array. If Memchr finds the byte in the array, it will return a pointer to the value. If it cannot find the byte, bret will be Null. Line 13 checks the return value to make sure that bret is not Null; using a Null pointer will cause the program to behave strangely or crash.

Line 14 calculates the array index of the found byte. Bret contains the address of byte 100 in the array. @aArray(1) returns the address of the first element of the array. Since arrays are sequential memory locations, you can find the array index by subtracting the address of the first array element from bret, and then adding the lower bound index to the value. This will give you the index of the array element. Line 15 prints the calculated index value to the console.

Line 23 copies the values from aArray to bArray. Again, the address of the first element of each array is passed to function, along with the number of bytes to copy. The number of bytes to copy is passed to the function with the code Ubound(aArray) * Sizeof(Ubyte), which uses the number of elements in the array times the size of a ubyte. Lines 24 through 28 then print the values of bArray to screen to verify that the array was in fact copied. The program is closed in the usual way.

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

```
Looking for byte 100...

100 found at index  3

Copying aArray to bArray...

bArray( 1 ) = 65
bArray( 2 ) = 123
bArray( 3 ) = 100
bArray( 4 ) = 52
bArray( 5 ) = 255
```

Output 12.5: Output of memfunc.bas

As you can see from the output, both operations succeeded. Memcopy is extremely handy when you need to make a copy of an array. The function is much faster than using a For-Next, and there considerably less code to write.

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.

A Look Ahead

Manipulating date and time values is a common task in programming. FreeBasic has a number of date and time functions that make working with date and time information quite easy, as you will see in the next chapter.

13 Date and Time Functions

Working with dates and time in many Basic languages can be problematic due to the lack of date and time functions. FreeBasic however, has a rich set of date and time functions, and when used in combination with the Format function, which you will see in the next chapter, you can easily handle any date or time problem you may encounter.

Intrinsic Date and Time Functions

FreeBasic includes the standard date and time functions that you see in most Basic languages. The following table lists these functions.

Function	Syntax	Comment
Date	B = Date	Returns the current system date as a string.
Setdate	B = Setdate(date_string)	Sets the system date to date_string, returning 0 on success and non-zro on failure. Date string must be in one of the following formats: mm-dd-yy", "mm-dd-yyyy", "mm/dd/yy", or "mm/dd/yyyy".
Settime	B = Settime(time_string)	Sets the system time to time_string, returning 0 on success and non-zero on failure. Time string must be in one of the following formats: "hh:mm:ss", "hh:mm", or "hh".
Time	B = Time	Returns the current system time as a string.
Timer	B = Timer	Returns the number of seconds since the computer was started as a double-type value. If the computer's CPU supports the Performance Counter, than Timer has a resolution in microseconds. On computers without the Performance Counter, the resolution is 1/18 of a second.

Table 13.1: Intrinsic Date and Time Functions

The following short programs demonstrates the return values for Date and Time as well as using Timer in a delay loop.

```
1    Option Explicit
2
3    Dim myTimer As Double
4
5    Print "Current time is ";Time
6    Print "Current date is ";Date
7    Print "Number of seconds since computer started: ";Timer
8    Print "Pausing for 2 seconds..."
9    myTimer = Timer + 2
10   Do
11       Sleep 10
12   Loop Until Timer > myTimer
13   Print "Press any key to continue..."
14
15   Sleep
16   End
```

Listing 13.1: datetime.bas

`Analysis:` Line 3 dimensions a double-type variable for use with the Timer function. Line 5 prints out the current system date. Line 6 prints out the current system time. Line 7 prints out the number of seconds since the computer was started. Line 9 initializes the variable myTimer with the current Timer result plus 2 seconds. The Do-Until loop in lines 10 through 12 loop until the 2 seconds have expired. The Sleep in line 11 allows the operating system to handle any other events while the computer is busy in the Do loop. Whenever you have an extended loop of this nature, you need to use a Sleep x command, where x is the number of milliseconds to wait. This will keep your program from consuming 100% CPU time and allow other processes to execute. The program is ended in the usual way.

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

```
Current time is 11:44:02
Current date is 06-27-2006
Number of seconds since computer started:  10710.42349851727
Pausing for 2 seconds...
Press any key to continue...
```

Output 13.1: Output of datetime.bas

As the example program illustrates, you can use the Timer function for delay loops. This is common in games that want to control the FPS, or frames-per-second, when using real-time animation techniques. In the code above, the initial Timer value is saved in a variable. Timer returns the number of seconds since the computer was started, so adding 2 to this variable will create a 2 second interval. The Until portion of the loop checks to see if the current result of the Timer function is greater than the saved value, and exits when 2 seconds have elapsed.

Extended Date and Time Functions

In addition to the functions listed above, FreeBasic has several extended date and time functions that enable you to easily calculate date and time intervals, and extract different parts of a date or time number for use in your program. In order to use these functions you must include `"vbcompat.bi"` in your program. Most of these functions require a date serial, a double-type value that represents a date and time number. The date is stored in the integer portion of the double, while the time is stored in the fractional part of the double.

DateAdd Function

The DateAdd function returns a future or past date or time based on a certain interval. The function requires a serial start date or time, an interval string and the number of intervals.

```
      date_serial = DateAdd(interval_string, number_of_intervals, 
start_date_serial)
```

The following list explains each of the parameters in order; that is, interval_string is 1 below, number_of_intervals is listed as 2, and so on, along with the return value.

1. Interval string: This is a string that indicates what interval to add to the start date_serial.
   a) “yyyy”: years
   b) “q”: quarters
   c) “m”: months
   d) “ww”: weeks
   e) “d”,”w”,”y”: days
   f) “h”: hours
   g) “n”: minutes
   h) “s:” seconds
2. Number of intervals: The number of intervals defined in interval_string to add to the passed date_serial.
3. Date serial: The starting date in the calculation.

4. Return: Returns a date serial that is offset by intervals from the starting date.

   The following program illustrates the DateAdd function.

```
Option Explicit

#Include Once "vbcompat.bi"

'Date serial variable
Dim nDate As Double
Dim interval As Integer = 2
Dim intervalstr As String

Print "Number of intervals:";interval
Print
Print "Current Date: ";Format(Now, "mm-dd-yyyy")
Print
'Year interval
intervalstr = "yyyy"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
'Quarter interval
intervalstr = "q"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
'Month interval
intervalstr = "m"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
'Week interval
intervalstr = "ww"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
'Day interval
intervalstr = "d"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
'w day interval
intervalstr = "w"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
'y day interval
intervalstr = "y"
nDate = DateAdd(intervalstr, interval, Now)
Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
```

```
42  'Quarter interval
43  intervalstr = "q"
44  nDate = DateAdd(intervalstr, interval, Now)
45  Print intervalstr;": ";Format(nDate, "mm-dd-yyyy")
46  Print
47  Print "Current time: ";Format(Now, "hh:mm:ss")
48  Print
49  'Hour interval
50  intervalstr = "h"
51  nDate = DateAdd(intervalstr, interval, Now)
52  Print intervalstr;": ";Format(nDate, "hh:mm:ss")
53  'Minute interval
54  intervalstr = "n"
55  nDate = DateAdd(intervalstr, interval, Now)
56  Print intervalstr;": ";Format(nDate, "hh:mm:ss")
57  'Second interval
58  intervalstr = "s"
59  nDate = DateAdd(intervalstr, interval, Now)
60  Print intervalstr;": ";Format(nDate, "hh:mm:ss")
61
62  Sleep
63  End
```

Listing 13.2: dateadd.bas

`Analysis:` In line 3 the “vbcompat.bi” is included in the program. You need to include this file if you want to use the extended date time functions as well as the format function. Line 6 dimensions a double-type variable to hold the return value from the DtaeAdd function. Line 7 creates an integer variable to hold the interval, which is set to 2. Line 8 creates a string that will hold the various interval strings. Line 10 prints the current interval number. Line 12 prints the current date using the Format function and the Now function. The Now function returns the current date and time as a date serial. The format function's format string “mm-dd-yyyy” will print a two digit month, a two digit day and a four digit year.

Lines 15 through 17 set the interval string for the DateAdd function, “yyyy” in this case, passes the interval string, interval and the current date serial, using the Now function. The calculated date is returned in nDate, which is passed to the Format function in line 17. Lines 18 through 45 use the same code blocks to calculate the result for each interval string.

Line 47 pints the current time. DateAdd works with time numbers as well as dates, and lines 50 through 60 illustrate the different time offsets. The Format function uses the format string “hh:mm:ss” in 47 which prints a 24-hour time. The same code blocks are used for the time calculations that were used in the date calculations, and the results are printed in the same fashion.

When you run the program your output should look similar to the following.

```
Number of intervals: 2

Current Date: 06-27-2006

yyyy: 06-27-2008
q: 12-27-2006
m: 08-27-2006
ww: 07-11-2006
d: 06-29-2006
w: 06-29-2006
y: 06-29-2006
q: 12-27-2006

Current time: 16:28:12

h: 18:28:12
n: 16:30:12
s: 16:28:14
```

Output 13.2: Output of datedd.bas

Using a negative interval number will return a past date, rather than a future date. Changing the variable interval in the above program to -2 will result in the following output. Notice that the date and time return values are two intervals in the past.

```
Number of intervals:-2

Current Date: 06-27-2006

yyyy: 06-27-2004
q: 12-27-2005
m: 04-27-2006
ww: 06-13-2006
d: 06-25-2006
w: 06-25-2006
y: 06-25-2006
q: 12-27-2005

Current time: 16:52:09
```

```
h: 14:52:09
n: 16:50:09
s: 16:52:07
```

Output 13.3: Output with Negative Interval Number

DateDiff Function

The DateDiff functions returns the difference between two date or time serials based on a certain interval. The function requires two date or time serials and an interval string. You can optionally specify the first day of the week or the first day of the year.

```
      int_value = DateDiff(interval_string, date_serial1, date_serial2)
      int_value = DateDiff(interval_string, date_serial1, date_serial2, 
first_day_of_week)
      int_value = DateDiff(interval_string, date_serial1, date_serial2, 
first_day_of_week, first_day_of_year)
```

You can specify the first day of the week when using the “ww” interval but the default is to use the system setting. You can also specify which week belongs to which year when a week spans the end of one year and the beginning of the next year.

1. Interval string: The interval strings are the same strings used in the DateAdd function.
2. Date Serial 1: The first date or time of the difference.
3. Date Serial 2: The second date or time of the difference.
4. First Day of the week: Specifies the day number of the first day of the week. This is used when using the “ww”, or week, interval string.
    a) Not specified: Sunday.
    b) 0: Local settings.
    c) 1: Sunday.
    d) 2: Monday.
    e) 3: Tuesday.
    f) 4: Wednesday.
    g) 5: Thursday.
    h) 6: Friday.
    i) 7: Saturday.
5. First day of the year: Specifies what year a week will be associated with when the week spans years.
    a) 0: Local settings.
    b) 1: January 1 week.
    c) 2: First week having four days in the year.

d) 3: First full week of year.

6. Return: Returns an integer value that represents the number of intervals between the two date or time values.

The following program illustrates the DateDiff function.

```
1   Option Explicit
2
3   #Include Once "vbcompat.bi"
4
5   Dim As Double Date1, Date2
6
7   'Get current date
8   Date1 = Now
9   'Get past date
10  Date2 = DateValue("1/1/2006")
11
12  'Print out number of days since first of year '06
13  Print "Number of days since ";Format(Date2, "mm/dd/yyyy");" is";DateDiff("d", Date2,
    Date1)
14
15  Sleep
16  End
```

Listing 13.3: datediff.bas

`Analysis:` Line 3 includes the extended date and time functions contained in the vbcompat.bi include file. Line 5 dimensions two double-type variables to hold the current and past date. Line 8 gets the current date and time. Line 10 uses the DateValue function to convert the string date to a DateSerial value. Line thirteen prints the number of days that have elapsed since January 1, 2006.

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

```
Number of days since 01/01/2006 is 225
```

Output 13.4: Output of datediff.bas

DatePart Function

The DatePart function returns a specific part of a DateSerial, using the specified interval. The function requires a DateSerial, the interval to return and optionally, the first day of the week and the first day of the year. The function returns an integer value representing the specified interval.

```
int_value = DatePart(interval_string, date_serial)
```

```
      int_value = DatePart(interval_string, date_serial, first_day_of_week)
      int_value = DatePart(interval_string, date_serial, first_day_of_week,
first_day_of_year)
```

1. Interval string: The interval strings are the same strings used in the DateAdd function.
2. Date Serial: A date or time serial number.
3. First Day of the week: Specifies the day number of the first day of the week. The values are the same as those listed in the DateDiff function.
4. First day of the year: Specifies what year a week will be associated with when the week spans years. The values are the same as those listed in the DateDiff function.
5. Return: Returns an integer representing the interval value.

The following program illustrates using the DatePart function.

```
1    Option Explicit
2
3    #Include Once "vbcompat.bi"
4
5    Dim As Double myDate
6
7    'Get current date
8    myDate = Now
9
10   'Print date values
11   Print "Day number is";Datepart("d", myDate)
12   Print "Week number is";Datepart("ww", myDate)
13   Print "Month number is";Datepart("m", myDate)
14   'Print the current time numbers
15   Print "Hour is";DatePart("h", myDate)
16   Print "Minutes are";DatePart("n", myDate)
17   Print "Seconds are";DatePart("s", myDate)
18
19   Sleep
20   End
```

Listing 13.4: datepart.bas

`Analysis:` Line 3 includes the extended date and time functions. Line 5 creates a double-type varibale to hold the current date and time serial. Line 8 uses the Now function to return the current system date and time. Line 11 prints the day number. Line 12 prints the week number. Line 13 prints the month number. Line 15 prints the hour number. Line 16 prints the minute number. Line 17 prints the second number. The program is closed in the usual way.

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

```
Day number is 15
Week number is 33
Month number is 8
Hour is 9
Minutes are 52
Seconds are 58
```

Output 13.5: Output of datepart.bas

The values you see will depend on the current date and time setting of your computer.

DateSerial Function

The DateSerial function will return a double-type DateSerial value based on the passed year, month and day. The date value returned by DateSerial has no time component.

```
date_serial = DateSerial(year, month, day)
```

1. Year: The year component of the date.
2. Month: The month component of the date.
3. Day: The day component of the the date.
4. Return: Returns a DateSerial that is the value of the year, month and day components.

The following program illustrates using the DateSerial function.

```
Option Explicit

#Include Once "vbcompat.bi"

Dim As Double myDate

'Build a date serial using the Datepart function
myDate = DateSerial(Datepart("yyyy", Now), Datepart("m", Now), _
         Datepart("d", Now))
Print "Current date is ";format(myDate, "mm/dd/yyyy")

Sleep
End
```

Listing 13.5: dateserial.bas

`Analysis:` Line 3 includes the extended date and time functions in the program. Line 5 dimensions a double-type variable which will contain the date serial value. Lines 8 and 9 create a date serial using the Datepart function to extract the year, month and day components from the current system date. Line 10 uses the Format function to print the date to the console screen.

When you run the program you should see the current system date.

```
Current date is 08/15/2006
```

Output 13.6: Output of dateserial.bas

Of course it is a bit silly to use the DatePart function when you could simply use the Now function to return the current system date, but this program shows how both function could be used together in a program. In a real program, the year, month and day values would be probably be coming from an external source which would need to be combined to create a date serial that would then be used for a date calculation.

DateValue Function

The DateValue function takes a string as it argument and returns a date serial value. The date string must be in the same format as the regional settings of the computer.

```
date_serial = DateValue(date_string)
```

1. Date String: A date string in the current regional setting format. DateValue will only convert a date string, not a time string.
2. Return: A DateSerial representing the date string.

The following program illustrates using the DateValue function.

```
Option Explicit

#Include Once "vbcompat.bi"

Dim As Double myDate
Dim As String myDateStr

'Create a date string value
myDateStr = "08/15/2006"
'Get the serial date value
myDate = DateValue(myDateStr)
```

```
12   'Print the value
13   Print "Date string is ";Format(myDate, "mm/dd/yyyy")
14
15   Sleep
16   End
```

Listing 13.6: datevalue.bas

`Analysis:` Line 3 includes the extended date and time functions. Line 5 creates a double DateSerial variable that will contain the converted value. Line 6 defines a string that will hold the date string value. Line 9 initializes the string variable to a date format, using the US regional settings. This may have to be changed on your computer if you are using a different setting. Line 11 creates a DateSerial from the string value. Line 13 prints the value using the Format function.

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

```
Date string is 08/15/2006
```

Output 13.7: Output of datevalue.bas

If you run this program and get a strange date, then the regional settings on your computer are different than what is shown in this program. For example, if your setting has the year first, then you will need to change the string format to have the year first in the string, rather than last. The DateValue function, used in conjunction with the IsDate function which is covered later in this chapter, is an easy way to get a date from the user which then can be used for date calculations.

Day Function

The Day function will return the day number from a DateSerial. The Day function takes a single date serial value and returns an integer representing the day number.

```
int_value = Day(date_serial)
```

1. DateSerial: The date serial value.
2. Return: Integer value representing the day of the month.

The Day function returns the same value as DatePart with the day interval specified.

Hour Function

The Hour function returns the current hour number from a TimeSerial value. The Hour function takes a single TimeSerial value, or a DateSerial value with a time component and returns an integer that represents the hour.

```
int_value = Hour(time_serial)
```

1. TimeSerial: The time serial value.
2. Return: Integer value representing the hour of the day. If the PM specifier is added to the time value, the hour number returned will be in 24 hour format. That is, 1:00 PM will return 13.

The following program illustrates using the Hour function.

```
1    Option Explicit
2
3    #Include Once "vbcompat.bi"
4
5    Dim As Double myTime
6    Dim As String myTimeStr
7
8    'Create a date string value
9    myTimeStr = "1:10:23 pm"
10   'Get the serial date value
11   myTime = TimeValue(myTimeStr)
12   'Print the value
13   Print "Hour is";Hour(myTime)
14
15   Sleep
16   End
```

Listing 13.7: hour.bas

`Analysis:` Line 3 includes the extended date and time functions. Line 5 creates a double-type TimeSerial variable. Line 6 creates a string variable to hold the string time value. Line 9 initializes the string variable to a time string. Line 11 converts ythe time string to a TimeSerial using the TimeValue function. Line 13 prints the hour number of the TimeSerial.

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

```
Hour is 13
```

Output 13.8: Output of hour.bas

Minute Function

The Minute function returns the minute number from a TimeSerial value. The Minute function takes a single TimeSerial value, or a DateSerial with a time component, and returns an integer representing the minute.

```
int_value = Minute(time_serial)
```

1. TimeSerial: The time serial value.
2. Return: Integer value representing the minute of the hour.

Month Function

The Month function returns the month number from a DateSerial value. The Month function takes a single DateSerial value and returns an integer representing the month.

```
int_value = Month(date_serial)
```

1. DateSerial: The date serial value.
2. Return: Integer value representing the month of the year.

MonthName Function

The MonthName function returns a string representing the name of the month. The MonthName function takes an integer representing the month, and an optional flag that will return a month abbreviation if set.

```
string_value = MonthName(month_number)
string_value = MonthName(month_number, abbreviate)
```

1. Month Number: An integer representing the month.
2. Abbreviate: A flag that indicates whether to return an abbreviated month name. If flag is 0, the full month will be returned. If the flag is 1, the abbreviated month name will be returned.
3. Return: A string value representing the name of the month. The return value is based on the current locale settings of the system.

The following program illustrates using the MonthName function.

```
Option Explicit

#Include Once "vbcompat.bi"

Dim As Integer i

For i = 1 To 12
```

```
8       Print "Month ";Monthname(i);" is abbreviated as ";Monthname(i, 1)
9   Next
10
11  Sleep
12  End
```

Listing 13.8: monthname.bas

`Analysis:` Line 3 includes the extended date and time function. Line 5 dimensions an integer variable for the following For-Next routine. The For-Next block in lines 7 through 9 call the MonthName function to get the full name, and then is called a second time with the abbreviate flag set to 1 to get month abbreviation.

When you run the program you should see output similar to the following, depending on your local settings.

```
Month January is abbreviated as Jan
Month February is abbreviated as Feb
Month March is abbreviated as Mar
Month April is abbreviated as Apr
Month May is abbreviated as May
Month June is abbreviated as Jun
Month July is abbreviated as Jul
Month August is abbreviated as Aug
Month September is abbreviated as Sep
Month October is abbreviated as Oct
Month November is abbreviated as Nov
Month December is abbreviated as Dec
```

Output 13.9: Output of monthname.bas

Now Function

The Now function returns a double-type value representing the current system date and time. The Now function takes no parameters.

```
double_value = Now
```

1. Return: A double value representing the current system date and time.

Second Function

The Second function returns an integer value representing the second number of a TimeSerial value, or a DateSerial with a time component. The Second function takes a single TimeSerial value.

```
int_value = Second(time_serial)
```

3. TimeSerial: The time serial value.
4. Return: Integer value representing the second of the minute.

TimeSerial Function

The TimeSerial function will return a double-type TimeSerial value based on the passed hour, minute and secondy. The time value returned by TimeSerial has no date component.

```
time_serial = TimeSerial(hour, minute, second)
```

1. Hour: The hour component of the time. The hour is passed in twenty-four format.
2. Minute: The minute component of the time.
3. Second: The second component of the the time.
4. Return: Returns a TimeSerial that is the value of the hour, minute and second components.

The TimeSerial function works just like the DateSerial function, except that it returns a time value rather than a date value. You can get both a date and time by adding a DateSerial value created with the DateSerial function to a TimeSerial value created with the TimeSerial function. Since the date is contained within the integer portion of the DateSerial and the time is contained within the fractional portion of the DateSerial, you can combine the two values by simply adding them. The following program illustrates this concept.

```
Option Explicit

#Include Once "vbcompat.bi"

Dim As Double ts, ds, cs

'Get a date serial
ds = Dateserial(2006, 1, 10)
'Get a time serial
ts = Timeserial(13, 10, 42)
'Combine the two
cs = ds + ts
'Print values
Print "Date and time is ";Format(cs, "mm/dd/yyyy hh:mm:ss AM/PM")

Sleep
End
```

Listing 13.9: timedateserial.bas

`Analysis:` Line 3 includes the extended date and time functions. Line 5 creates three double-type values, ts which will be the time serial value, ds which will be the date serial value and cs, which will be the combined date and time serial value. Line 8 gets the date serial value using the DateSerial function. Line 10 gets the time serial value from the TimeSerial function. Line 12 adds the two values together to get a combined date and time serial value. Line 14 uses the Format function to print both the date and time. The AM/PM specifier is added to the time format string to convert the 24-hour value to a 12-hour value.

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

```
Date and time is 01/10/2006 01:10:42 PM
```

Output 13.10: Output of timedateserial.bas

TimeValue Function

The time value function will return a TimeSerial value from a string representing the time value. The time string can contain either a 24-hour format such as “13:10:42” or a 12-hour format such as “01:10:42PM”.

```
time_serial = TimeValue(time_string)
```

1. Time String: A time string in either 24 or 12-hour formats.
2. Return: A TimeSerial representing the time string.

Year Function

The Year function returns an integer value representing the year number of a DateSerial value. The Year function takes a single DateSerial value.

```
int_value = Year(date_serial)
```

1. DateSerial: The date serial value.
2. Return: Integer value representing the year of the date serial value.

Weekday Function

The Weekday function returns the number of the day of the week. The Weekday function takes a DateSerial value and an optional flag that represents the first day of the week.

```
int_value = Weekday(date_serial)
int_value = Weekday(date_serial, first_day_of_week)
```

1. DateSerial: The date serial value.
2. First Day of Week: A flag that represents the day number that is regarded as the first day of the week. The values are the same as those in the DateDiff function.
3. Return: Integer value representing the day of the week.

WeekDayName Function

The WeekDayName function returns a string that is the name of the day. The WeekDayName function takes an integer representing the day of the week, an optional flag that indicates whether to return an abbreviated name and an option flag that indicates the first day of the week.

```
int_value = WeekDayName(day_number)
int_value = WeekDayName(day_number, abbreviate)
int_value = WeekDayName(day_number, abbreviate, first_day_of_week)
```

1. Day Number: The day number. The day number is interpreted by the first-day-of-the-week setting.
2. Abbreviate: If this flag is 1, the function will return an abbreviated name, if 0 then the full name is returned.
3. First Day of Week: A flag that represents the day number that is regarded as the first day of the week. The values are the same as those in the DateDiff function.
4. Return: A string value that represents the name of the day of the week. The day name is based on the computer's locale setting.

The function works juts like the MonthName function with the added first-day-of-the-week setting.

A Look Ahead

Once you have created the data in your program, whether it be numeric, date or time data, you will at some point need to format that data. The Format function has all the capability you need to format these data types, and is the subject of the next chapter.

14 The Format Function

The Format function is a general purpose formatting function that can handle a wide range of numeric values. Once you understand the capabilities of the function, you will find yourself using this function for all your formatting routines. The Format function can handle simple numeric data, date and time values.

```
string_value = Format(numeric_value, format_string)
```

The numeric value parameter can any of the numeric values available in FreeBasic. The format string contains format characters that specify how to format the numeric value. If the Format string is an empty string, then Format behaves just like the Str function.

String Format Characters

You can display text within the format string using the general formatting string. The following table lists the string format characters.

Character	Comment
: ? + $ () space	Will return as a literal string.
\	Will return the next character in the string as a literal character.
“text”	Will return text within double quotes as a literal string.
:	Used as a separator for time components.
/	Used as a separator for date components.

Table 14.1: String Format Characters

The values listed in the table will return as literal text, and can be used to generate a string that contains both literal characters and formatted numeric text.

Numeric Format Characters

There are a number of format characters that can be used to format simple numeric types. The following table lists the format characters.

Character	Comment
0 (zero)	The zero is a digit placeholder. How the number is formatted depends on how many zeros are in the format string and how many numbers on either side of the

Character	Comment
	decimal point of the number. If there are fewer digits than 0's, a leading or trailing zero are added to the format string. If there are more digits to the right of the decimal than there are 0's, then the number is rounded. The number to the left of the decimal are not changed.
#	This format character works in the same way as the 0 character, except the leading or trailing zeros are not displayed.
. (decimal point)	Will return a decimal point within the formated string. If the format string only has #'s to the left of the decimal point, then numbers smaller than 1 will be displayed with only a leading decimal point.
%	Will multiply number by 100 and return percentage sign.
, (comma)	The comma is used as the thousands separator. Two adjacent commas, or a comma right after the decimal point, will not display the thousands digits and will round the number.
E- E+ e- e+	These characters are used to format a string in scientific format. Placing a 0 or # to either the left or right of the character determines the number of digits to display on either side of the exponent sign. The negative characters are used to display negative exponents and the positive characters are used to display positive exponents.

Table 14.2: Numeric Format Characters

As you can see from the character table, there is a wide range of formatting options available for formatting numeric data. The following program demonstrates how to use some of the formatting options.

```
Option Explicit

#Include Once "vbcompat.bi"

Dim As Double myValue = 123456.789

Print "Value: ";Format(myValue)
'Print various format string
```

```
9    Print "Value as currency: ";Format(myValue, "$###,###,###.##")
10   myValue = .10
11   Print "Value: ";Format(myValue)
12   Print "Value as scientific: ";Format(myValue, "###E+###")
13   Print "Value with zeros: ";Format(myValue, "00.0000")
14   Print "Value as percentage: ";Format(myValue, "##%")
15
16   Sleep
17   End
```

Listing 14.1: format-numeric.bas

`Analysis:` Line 3 includes the extended date and time function library which also includes the Format function. Line 5 creates a double value that will be used as the numeric value in the formatting expressions. Line 7 prints the unformatted value. Line 9 prints the value as a currency value using the $ with commas used to separate the thousand places and two # characters after the decimal point to round the value. Line 10 sets the double variable to .1 for different formatting. Line 11 prints the unformatted value. Line 12 prints the value in scientific format. Line 13 prints the value with leading and trailing zeros. Line 14 prints the value as a percentage.

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

```
Value: 123456.789
Value as currency: $123,456.79
Value: .1
Value as scientific: 100E-3
Value with zeros: 00.1000
Value as percentage: 10%
```

Output 14.1: Output of format-numeric.bas

Date Format Characters

In the previous chapter you saw some of the date format characters used in the example programs. The following table lists all the date format Characters and their meaning.

Character	Comment
d	Returns the day of the month without leading zeros.
dd	Returns the day of the month with leading zeros.
ddd	Returns the day of the week name as an abbreviation.

Character	Comment
dddd	Returns the day of the week name as a full name.
ddddd	Returns the serial date formatted according the locale setting. Includes the year, month and day.
m	Returns the month number without leading zeros.
mm	Returns the month number with leading zeros for single month numbers.
mmm	Returns the month name as an abbreviation.
mmmm	Returns the month name as a full name.
y	Returns the year number formatted as two digits.
yyyy	Returns the year number formatted as four digits.

Table 14.3: Date Format Characters

The following program demonstrates some of the formatting options for dates.

```
1    Option Explicit
2
3    #Include Once "vbcompat.bi"
4
5    Dim As Double myDate = Now
6
7    'Print some formatted dates
8    Print "Date Serial: ";Format(myDate, "ddddd")
9    Print "Date with single chars: ";Format(myDate, "d/m/yy")
10   Print "Date with double chars: ";Format(myDate, "dd/mm/yyyy")
11   Print "Date with year first: ";Format(myDate, "yyyy/mm/dd")
12   Print "Date with (-): ";Format(myDate, "dd-mm-yyyy")
13   Print "Date with ( ): ";Format(myDate, "dd mm yyyy")
14
15   Sleep
16   End
```

Listing 14.2: format-date.bas

`Analysis:` Line 3 includes the date-time and format library definition. Line 5 creates a date serial values and initializes it to the current date and time using the Now function. Line 8 prints the serial date using the local settings. Line 9 prints the date as single month and day values with a two-digit year. Line 10 prints the date with two-digit month and day values and a four-digit year. Line 11 prints the date with the year first. Line 12 using a dash to separate the date components and line 13 uses a space to separate the date components.

When you run the program you should see the following.

```
Date Serial: 8/16/2006
Date with single chars: 16/8/06
Date with double chars: 16/08/2006
Date with year first: 2006/08/16
Date with (-): 16-08-2006
Date with ( ): 16 08 2006
```

Output 14.2: Output of format-date.bas

Time Format Characters

The time format characters are similar to the date format characters and are used in much the same way. The following table lists the format characters that you can use with a TimeSerial value.

Character	Comment
h	Returns the hour without a leading zero.
hh	Returns the hour with a leading zero for single digit hour numbers.
m	Returns the minute without a leading zero.
mm	Returns the minute with a leading zero for single digit minute numbers.
s	Returns the second without a leading zero.
ss	Returns the second with a leading zero for single digit second numbers.
ttttt	Returns a TimeSerial value formatted as the local specific time format.
AM/PM am/pm	Converts the time to a 12-hour format and adds an am or pm to the return value.
A/P a/p	Converts the time to a 12-hour format ans adds an a or p the return value.

Table 14.4: Time Format Characters

The following program demonstrates some ways to format a time value.

```
1    Option Explicit
2
3    #Include Once "vbcompat.bi"
4
5    Dim As Double myTime = Timeserial(8, 1, 1)
6
7    'Print some formatted times
8    Print "Time Serial: ";Format(myTime, "ttttt")
9    Print "Time with single chars: ";Format(myTime, "h:m:s")
10   Print "Time with double chars: ";Format(myTime, "hh:mm:ss")
11   Print "Time with am/pm: ";Format(myTime, "hh:mm:ss am/pm")
12   Print "Time with a/p: ";Format(myTime, "hh:mm:ss a/p")
13
14   Sleep
15   End
```

Listing 14.3: format-time.bas

`Analysis:` Line 3 includes the extended date and time library. Line 5 creates a time serial value using the TimeSerial function. Line 8 prints the time using the TimeSerial format. Line 9 prints the time using single-digit characters. Line 10 prints the time using two-digit characters. Line 11 prints the time as in 12-hour format using the AM/PM format, and line 12 also prints the time in a 12-hour format with the a/p format.

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

```
Time Serial: 8:01:01 AM
Time with single chars: 8:1:1
Time with double chars: 08:01:01
Time with am/pm: 08:01:01 am
Time with a/p: 08:01:01 a
```

Output 14.3: Output of format-time.bas

The Format function has a tremendous amount of functionality, and when working with numeric data, it is an invaluable resource in formatting. You should some time to become acquainted with this function as it will save you lot of time when you need to format your data.

A Look Ahead

At this point you should have a good understanding of the data types available in FreeBasic. Creating data is only half the story though. You need to display that data and you need to gather data from your user. The next chapter shows how to make your screen output look interesting and explains the various ways to gather user input.

15 Console Programming

Console programming, or more generally text-based programming, is a programming style that uses text as the primary medium to display information to, and to interact with, the user. This is in contrast to GUI programming, where a graphical user interface is used for display and interaction or a graphics-based program where graphical objects are used for display and interaction. Text-based programming has its roots in the early days of computing, when computers did not have graphical displays or the graphics were too primitive to be of use. Even today, in this era of photo-realistic graphics, text-based programming remain a vibrant area of application development.

At first glance it may seem that text-based applications would be limited, however this is not the case at all. Over the years everything from sophisticated word processors and spreadsheets to games have been developed using only text. Text user interfaces, or TUIs have been developed that have all the functionality of modern graphical user interfaces. Many of the most popular games in history, such as Zork, were written entirely in text. Text-based application are content oriented, and if conveying content is what you are trying to do, then a text-based application is well suited to the task.

If you are interested in text-based programming you should visit Ascii World located at http://www.ascii-world.com. Ascii World is a site devoted entirely to text-based programming, regardless of the programming language.

The Console Screen

The console can be viewed as a grid, where each row contains a certain number columns. In FreeBasic you can specify how many rows and columns a console can contain, providing the operating system supports the configuration. The basic console configuration is 25 rows by 80 columns. Consoles are row oriented, so you specify a position on the grid using the row followed by the column Each grid location can contain a single character.

The character set for the console under Windows®, is the Ascii character set. The Ascii character set contains 256 characters and includes all the numbers, letters and punctuation you would expect, as well as some extended characters used in European languages, and a set graphical characters that can used for drawing lines and boxes on the screen. By using both text and the Ascii graphical characters in a console application you can create quite sophisticated user interfaces completely in text.

The Console Functions

FreeBasic has a number of console functions that enable you to position text on the screen, set the foreground and background colors of the text, and to retrieve information from the console screen. You can also use the mouse in the console for added functionality. The following table lists the console commands available in FreeBasic.

Function	Syntax	Comment
Cls	Cls mode	Clears the screen. If mode is

Function	Syntax	Comment
		0 or omitted, entire screen is cleared. If mode is 1 the graphics viewport is cleared. If mode is 2, the text viewport is cleared.
Color	Color foreground, background B = Color	Sets the foreground and/or background color of printed text. Color can also be used to return the current color attributes, where the foreground color is the low-word and the background color is the high-word.
Csrlin	B = Csrlin	Returns the row that the cursor currently occupies.
Format	B = Format (numeric, format_string)	Formats a numerical expression according to format_string.
Getmouse	Getmouse col, row, wheel, button	Returns the current position of the mouse cursor in row, col, the current wheel state and button state.
Locate	Locate row, column, cursor_state	Positions the cursor on the row and column. If cursor_state is 0, cursor is hidden. If 1, cursor is shown.
Pos	B = Pos	Returns the current column of the cursor.
Print ?	Print expression Print expression, Print expression; ? expression ? expression, ? expression;	Prints expression to the screen. Expression can be text, a string variable or numeric variable. A comma at the end of the print string will print a tab space with no carriage return. A semi-colon will not print a carriage return. If no comma or semi-colon, a carriage return is printed.
Print Using	Print Using format_string, expression1 Print Using format_string, expression1, Print Using format_string, expression1;	Print expression to the screen using a format string. Print Using has some known bugs and should not be used. Use format instead.

Function	Syntax	Comment
Screen	B = Screen(row, col, color_flag)	Screen returns the ascii character code at row and column position if color_flag is 0. If color_flag is 1, Screen returns the color attribute. You can determine the background color with `background = attribute shr 4` and the foreground color with `foreground = attribute and &hf`.
Spc	Spc columns	Causes next print statement to skip number of columns.
Tab	Tab column	Sets column for next print statement.
View Print	View Print first_row To last_row	Sets printing area of screen.
Width	Width columns, rows	Sets the the number of columns and rows of the console window.

Table 15.1: Console Commands

Console Colors

The default color scheme for the console is white text on a black background. You can change the color scheme by using the Color function to change either or both the foreground and background color of printed text. For the console, the color numbers range from 0, black, to 15, bright white. The following list enumerates the color numbers that can be used in the console.

- 0: black
- 1: blue
- 2: green
- 3: cyan
- 4: red
- 5: magenta
- 6: yellow
- 7: white
- 8: gray
- 9: bright blue
- 10: bright green
- 11: bright cyan

- 12: bright red
- 13: bright magenta
- 14: bright yellow
- 15: bright white

The colors listed above may vary, depending on your graphics card and monitor, but they should approximate those listed. In addition to printing colored text, you can also set the background of the entire console screen by setting the background color using the Color function, and then using Cls to clear the screen. When the screen is cleared, the background will be in the selected color.

Color is useful for enhancing the look of your application, but you should never rely on color alone to indicate critical information in your application. Approximately 15 million people are color blind just in the United States. Depending on the level of color blindness, an individual may miss critical information if it is displayed using color alone. It is good practice to identify critical information through the use of text manipulation, such as a pop-up box, so that the information won't be missed by a color blind user.

Each character position on the console screen stores the character value, along with the current color attributes which can be determined using the Screen function. If color_flag is 0, Screen will return the ascii code of the character at the selected row and column. If color_flag is 1, Screen will return the color attribute. Colors are stored as byte values with the high 4 bits indicate the background color and the low 4 bits indicate the foreground color, the color of the text. You can determine the color attributes by using the following formulas.

```
Foreground = attribute And 15
Background = attribute  Shr 4
```

The following program illustrates how you might use these functions in a program.

```
Option Explicit

Dim As Integer clr, ccode

'Set the color
Color 15, 0

'Get the current color
clr = Color

'Print foreground and background color numbers
Print "Current color setting:"
Print "Foreground: ";Loword(clr)
Print "Background: ";Hiword(clr)
```

```
15   Print
16   'Get the current character position at 1, 1
17   ccode = Screen(1, 1)
18   Print "Character code at 1,1 is";ccode;" which is ";Chr(ccode);"."
19
20   'Get the color attributes at 1,1
21   clr = Screen(1, 1, 1)
22   Print "Foreground color at 1,1:";clr And 15
23   Print "Background color at 1,1:";clr Shr 4
24
25
26   Sleep
27   End
```

Listing 15.1: color.bas

`Analysis:` Line 3 dimensions the working variables, clr which contains the color attribute of a text position and ccode which is the character code. Line 6 sets the current color to bright white, 15, with a black background, 0. Line 9 stores the current clr setting in the variable clr. Lines 13 and 14 print the current color numbers to the screen using the Loword and Hiword macros. Line 17 stores the character code at row position 1, column position 1 in ccode. Line 18 prints both the character code and the corresponding character to the screen. Line 21 retrieves the color attribute for the same position, and lines 22 and 23 print the color values using the formula listed above. The program is closed in the usual way.

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

```
Current color setting:
Foreground: 15
Background: 0

Character code at 1,1 is 67 which is C.
Foreground color at 1,1: 15
Background color at 1,1: 0
```

Output 15.1: Output of color.bas

The display will be in bright white, with a black background. As you can see, you have tremendous control over the appearance of console text and can easily determine the attributes of the text.

Positioning Text

You can position text anywhere on the console screen using the Locate statement. The Locate statement sets the text cursor position for subsequent printing of text. To print to a certain location on the screen, you must first position the text cursor using the locate statement, then follow that command with a print statement. You can turn the text cursor on and off by using the optional cursor flag of the locate statement. 1 turns the text cursor on, and 0 turns the cursor off. This is useful if, for example, you are using the mouse for console input.

The Print command will update the current cursor position. You can obtain the current row setting by using the Csrlin function and the current column setting by using the Pos function.

Printing Text

The Print statement is used to print text to the screen. The location and color of the text is determined by the current cursor setting and the last color statement. Using the Print statement with a semi-colon after the print string will position the text cursor to the end of the string. Using a comma after the print string will position the text cursor to the next tab location. A Print statement without either a semi-colon or comma, will insert a carriage return and/or line feed positioning the cursor at the beginning of the next line.

You can format the output of numeric text using the Print Using command, although this is a deprecated function and has some known, minor bugs. The function is available mainly for backward compatibility with older programs. For new programs, you should use the Format function, which is covered in a separate chapter later in this book. For details on the Print Using function see the FreeBasic documentation wiki[4]. The function will not be covered in this book.

You have already seen the Spc and Tab functions in several of the example programs. The Spc function will cause the next print position to skip a number of columns while the Tab function will put the text cursor at the next tab position. Both of these functions are useful for printing tabular data to the screen.

Determining and Setting the Size of the Console

You can determine the current console size, or set the number of rows and columns of the console by using the Width function. To determine the current console size, use the Width function without any parameters as shown in the following code snippet.

```
MyInt = Width
columns = Loword(myInt)
rows = Hiword(myInt)
```

The number of columns of the console is stored in the low word of the return integer value while the number of rows is stored in the high word of the integer. To set the console size, use the Width function with the number of desired columns and rows, as illustrated in the following code snippet.

```
Width 80, 25
```

This will set the console to 80 columns by 25 rows. The standard console configuration is 80 columns by 25 rows, however, if you need a different size for your

[4]At the time of this writing, the Print Using wiki page is loacted at http://www.freebasic.net/wiki/wikka.php?wakka=KeyPgPrintusing.

application you can use the Width function to check if the console is the appropriate size and then resize if necessary. Setting the console size will reset the cursor position to row 1 and column 1, so you may need to issue a Locate statement to reset the cursor position after the change.

The following program shows how to use the Width function to get the consile size information and reset the console.

```
1     Option Explicit
2
3     Dim As Integer consize, rows, cols
4
5     'Get the current console size
6     consize = Width
7     rows = Hiword(consize)
8     cols = Loword(consize)
9     Print "Current Rows =";rows
10    Print "Current Columns =";cols
11    Print "Press any key..."
12    Sleep
13    Cls
14
15    'Resize console if necessary
16    If rows > 25 Then
17        Width 80, 25
18    End If
19
20    'Get the new console size
21    consize = Width
22    rows = Hiword(consize)
23    cols = Loword(consize)
24    Print "New Rows =";rows
25    Print "New Columns =";cols
26
27    Sleep
28    End
```

Listing 15.2: width.bas

`Analysis:` Line 3 dimensions three variables, consize that will contain the Width return value, rows which will contain the number of rows and cols which will contain the number of columns. Line 6 gets the current console size. Line 7 retrieves the number of rows using the Hiword macro and line 8 retrieves the number of columns using the Loword macro. The current setting is printed to the screen in lines 9 and 10. The Sleep command is used in line 12 to pause the program and wait for a key press from the user. Line 13 clears the screen.

Lines 16 through 18 check to see if the number of rows is greater than 15 and if true, then resets the console to 80 columns by 25 rows. Line 21 checks the new console size and the current rows and columns are retrieved in lines 22 and 23. The new values are printed in lines 24 and 25. The program is then closed in the usual way.

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

```
Current Rows = 25
Current Columns = 80
Press any key...
```

Output 15.2: Output a of width.bas

```
New Rows = 25
New Columns = 80
```

Output 15.3: Output b of width.bas

Output a shows the console setting when the program starts, and Output b shows the console setting after checking the console size. In this run the default setting was the desired size, so the Width statement did not need to be executed.

Getting User Input

If you need to interact with the user, then you will need to get user input. There are several ways to collect user input which are listed in the following table.

Function	Syntax	Comment
Inkey	B = Inkey	Returns a key from the keyboard buffer as a string. If an extended key is pressed, the return string contains character 255 + key character. Inkey does not wait for keyboard input.
Input	Input variable_list Input “prompt”; variable_list Input “prompt”,variable_list	Input will gather input for one or more variables listed in variable_list. The variables should be separated with commas. The type of variable determines how input will collect the data. If the data type of the variable is a string, Input will convert the input data to a string. If the data type of the variable is

		numeric, Input will convert the data to a number. If an invalid numeric string is entered, Input will attempt to convert as many characters as possible to a number. If no characters can be converted to a string, Input will return 0. If a semi-colon is used with Input, a question mark will be displayed after the prompt. If a comma is used, a question mark will not be printed.
Input()	B = Input(number_of_characters)	Input() will return a string containing number of characters entered.
Getkey	B = GetKey	Returns the ascii code of the current key in the keyboard buffer. Getkey will wait for a key. If an extended key is pressed, the first character is skipped, character 255, and the second character is returned.

Line Input	Line Input string_variable
	B = Line Input "prompt"; string_variable
	B = Line Input "prompt", string_variable
		Line Input will return a line of text from the keyboard. All characters up to, but not including the enter key are read. An optional prompt can be displayed with either a question mark, using a semi-colon, or no question mark when using a comma.

Winput()	B = Winput(number_of_characters)	Wide string version if Input().

Table 15.2: Input Functions

Inkey

Inkey is useful for polling the keyboard buffer in a loop and can be used in games or in text-based user interfaces since Inkey does not with for a key to be pressed. This allows you to process other events within your program, and then take action if a key has been pressed. If the buffer is empty, Inkey will return an empty string. If a key is waiting in the buffer, Inkey will return a string that is either one or two character long. Keys can be either the normal printable keys, such as A or 1, or an extended key such as the Up Arrow or F1. Normal keys return 1 key code, while the extended keys return 2 key codes, with the first code being character 255.

The following program will display the character and ascii code(s) of both normal and extended keys.

```
1     Option Explicit
2
3     Dim As String key, char
4     Dim As Integer i
5
6     Do
7         'Check keyboard buffer
8         key = Inkey
9         'If key has been pressed
10        If Len(key) > 0 Then
11            'Clear the last message
12            Locate 1, 1
13            Print Space(79);
14            'Reposition cursor
15            locate 1,1
16            'Print current key codes
17            For i = 1 To Len(key)
18                char = Mid(key, i, 1)
19                Print "Character: ";char; " Code: ";Asc(char); " ";
20            Next
21        End If
22        Sleep 10
23    Loop Until key = Chr(27)
24
25    End
```

Listing 15.3: inkey.bas

Analysis: Line 3 dimensions two string variables, key which will be the return value from Inkey and char which will be the individual characters. The integer variable i in line 4 will be used in the For-Next to extract the individual characters fro the key variable. Line 6 through 23 define a keyboard processing loop using. Line 8 checks for keyboard input. Line 10 checks to see if the length of key is greater than zero, indicating that Inkey returned a key. Lines 12 through 15 clear the previous output line that may have been printed. Lines 17 through 20 extract each character from the key variable, loading the actual character into the variable char in line 18 and then printing the corresponding ascii code to the screen in line 19.

The Sleep command is used in line 22 to allow the operating system to process other events. If you are using an extended loop such as this keyboard input loop, you should always insert a Sleep # command in the loop to allow the operating system to process background tasks, otherwise your program may consume 100 percent of the operating system's time.

Line 23 then checks to see if the escape key was pressed, which is ascii code 27, and then exits the loop if the key was pressed. The program is then closed using the end statement.

A sample output is shown below.

```
Character:   Code: 255 Character: ; Code: 59
```

Output 15.4: Output of inkey.bas

The output shows the codes when the F1 key is pressed. Once you know the key codes, then you can build Const statements to trap for certain keys. For example, to trap for the F1 key, you would create the Const statement shown in the code snippet below.

```
Const F1_key = Chr(255) & Chr(59)
```

You can then check the Inkey return to see if the key returned natches the F1 key.

```
If key = key_F1 Then
        'Then do something
End If
```

You can set up any number of keys in this manner. The F1 key is usually used to display help content, and in your application you could display a help screen by executing the help display code in the body of the If statement. If you are coding a game, you might want to use the arrow keys to move your character player around the screen, by setting up the arrow key constants and then adding a If-Then-Elseif ladder to process the arrows keys.

Getkey

Unlike Inkey, Getkey does wait for the user to press a key. If an extended key is pressed, Getkey will discard the first character and return the second character. Getkey is used in situations where you are not interested in the extended key set and you need to pause the program to gather user input.

Input

The Input function is used to gather information from the user that will be loaded into one or more variables. The data type of the variable determines how the user input is processed. If the variable is a string, Input will process the information as a string; if the variable is a numeric data type, Input will try to convert the information to a number. The variables must be separated using commas.

You can display a prompt string with Input by enclosing a literal string before the variable list. If a semi-colon is used after prompt string, a question will be displayed after the prompt. If a comma is used after the prompt string, no question mark is printed.

If you are supplying multiple variables to Input, then the user will need to enter the values separated by either spaces or commas. The spaces and commas are not returned by Input. If you need to process either spaces or commas, then you need to use Line Input rather than Input.

Line Input

Line Input will return a single string up to but not including the enter key. That is, Line Input expects a single string variable, rather than multiple variables. Line Input will read both spaces and commas in the entered string. Like Input, you can use a semi-colon to display a question mark at the end of the prompt string. Unlike Input however, you can use either a literal string or a string variable for the prompt. Line Input returns the user input as a string and does not convert numeric strings to numbers.

Using the Mouse

Another way to get user input is to use the mouse. The Getmouse function will return the current zero-based row and column of the mouse as well as the state of the buttons and mouse wheel. Since the rows and columns are zero-based, you will need to add 1 to the values to map the return values to the screen

The current state of the mouse wheel as an integer that starts at zero when the program starts and will return either a more positive number if the mouse wheel is turned in a forward direction and a more negative number if the mouse wheel is turned in the reverse direction. Normally, you would store the mouse wheel value at program start, and then take an action based on the offset value returned by Getmouse. For example, to scroll some text, you may scroll up if the wheel number is moving in the positive direction and scroll down if the wheel number is moving in the negative direction.

You can also determine which buttons have been pressed by checking the button return value. Getmouse supports three buttons, left and right and a middle button, which often doubles as the mouse wheel. You can determine which button has been pressed by using the following formulas.

- Left button = button And 1
- Right button = button And 2
- Middle button = button And 4

Button is the value returned by the Getmouse function. Not only can you check for single button presses, but you can also check to see if both button have been pressed by checking each button state simultaneously. There is no way to check for a double click natively however, so you will need to create your own timing code to determine if a button has been double-clicked.

The following program illustrates using the Getmouse functions.

```
Option Explicit

'Mouse return values
Dim As Integer mx, my, oldmx, oldmy, mwheel, mbuttons

'Turn off text cursor
```

```
7    Locate ,,0
8    Do
9        'Get mouse info
10       Getmouse mx, my, mwheel, mbuttons
11       'Zero based so add 1
12       mx += 1
13       my += 1
14       'Check to see if we have new information
15       If (mx > 0) And (my > 0) then
16           Locate 1, 1
17           'Print current row, column and wheel setting
18           Print "Row:";my;" Col:";mx;" W:";mwheel;" BT:";
19           'Check for button press
20           If mbuttons And 1 Then Print "L";
21           If mbuttons And 2 Then Print "R";
22           If mbuttons And 4 Then Print "M";
23           Print "        "
24           'Clear old cursor
25           Locate oldmy, oldmx
26           Print chr(32);
27           'Print new cursor
28           Locate my, mx
29           Print Chr(15);
30           'Save the old values
31           oldmx = mx
32           oldmy = my
33       End If
34       Sleep 10
35   Loop Until Inkey$<>""
36   'Turn on cursor
37   Locate ,, 1
38
39   End
```

Listing 15.4: con-mouse.bas

`Analysis:` Line 4 creates the mouse parameter working variables. Mx and my are the current column and row returned by the Getmouse function. Oldmx and oldmy are the old column and row settings that are used to erase the mouse cursor character that is displayed on the screen. Mwheel is the current mouse wheel setting while mbuttons is the current button state. Line 7 uses the Locate statement to turn off the text cursor. Line 8 through 35 are the mouse input loop that polls the mouse. Line 10 uses the Getmouse function to return the current mouse state information. Since Getmouse returns the row and column as a zero-based value, lines 12 and 13 add 1 to the value to map the return value to the actual screen coordinates. Line 15 checks to see if the mouse is still within the console screen. If the mouse is outside the console screen, Getmouse will return -1 for the row and column values. Since the program added 1 to these values, if the mouse is outside the console window, the mx and my values will be 0; that is, -1 + 1 = 0. You should always check to make sure the mouse is still inside the console window before interacting with the mouse.

Lines 16 and 18 prints the current row, column and wheel settings to the console screen. Lines 20 through 22 check to see what mouse button, if any, has been pressed and prints the mouse button id to the screen. L is the left mouse button, R is the right mouse button and M is the middle button. Lines 25 and 26 erase the old mouse location indicator, an asterisk that will follow the mouse cursor on the screen. Lines 28 and 29 then print the asterisk at the new mouse location. Lines 31 and 32 save the current mx and my values in oldmx and oldmy, to use in the next loop iteration.

The Do-Loop is exited in line 35 when any key is pressed. Line 27 turned the text cursor back on and the program is closed with the End statement in line 39.

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

```
Row: 2 Col: 4 W: 2 BT:L
     ☼
```

Output 15.5: Output of con-mouse.bas

If the mouse does not work in the console window, then you will need to turn off the Quick Edit mode of the console. Right click the title bar and select Properties. Under Edit Options, uncheck the QuickEdit mode. If a pop-up window appears, select Save Properties for Future Windows with Same Title, then click the OK button. The mouse should not work in the console window.

Creating a Text View Port

You can create a text view port using the View Print statement. A text view port can be thought of as a window -within-a-window, where all text operations occur. You define a view by passing the beginning row and ending row to the View Print command. Once the command has been issued, all subsequent text operations occur within the view port.

The following program creates a 10 line view port and prints some text within the view, which can be scrolled using the mouse wheel. If the mouse does not work within the console window, you will need to turn off the QuickEdit mode as described above.

```
Option Explicit

'Create paragraph
Dim Shared As String * 80 myText(1 To 19) => { _
"    Lorem ipsum dolor sit amet consectetuer id a ut magna Vestibulum.", _
"Id elit justo pretium commodo vitae convallis urna magnis mattis ", _
"tempor. Duis vitae id semper euismod dolor wisi turpis leo dolor in. ", _
"Neque non Vivamus ac eget nunc sagittis ut mauris at orci. Tincidunt ", _
"ipsum habitasse accumsan leo iaculis et vitae congue vitae malesuada. ", _
"Wisi sem Curabitur id ut ipsum adipiscing facilisi elit In id. Et urna.", _
"    Lorem ipsum dolor sit amet consectetuer eleifend Suspendisse sem ", _
"lacus iaculis. Wisi feugiat mus augue parturient et felis Pellentesque ", _
"wisi et nec. Vitae metus senectus a dolor Nullam amet accumsan integue ", _
"nonummy id. Sed justo in fames turpis justo platea ultrices est ", _
"convallis vel. Interdum ut Morbi enim ut Vestibulum senectus lacinia", _
"Sed congue metus.", _
"    Lorem ipsum dolor sit amet consectetuer lacinia molestie wisi", _
"semper pretium. Et gravida congue volutpat ac Maecenas elit Nunc ", _
"orci Curabitur lobortis. Felis tincidunt nonummy vitae at ut et ", _
"et montes pellentesque Donec. Vitae elit habitant nunc cursus ", _
"ullamcorper risus sagittis tempus consectetuer velit. Est ", _
"Vestibulum non id pulvinar Nam vitae metus id congue massa. ", _
"Phasellus."}

Dim As Integer idx = 1, mx, my, oldwheel, mwheel, delta
Dim As Integer vtop = 10, vbot = 20

Sub PrintText(tidx As Integer)
    Dim As Integer i, tbot

    'Clear view port to black
    Color , 0
    Cls
    'Calc text that will fit into view port
    tbot = tidx + 10
    If tbot > 19 Then tbot = 19
    For i = tidx To tbot
        Print Rtrim(myText(i))
    Next
End Sub

'Set the console size
Width 80, 25
'Turn off text cursor
Locate ,,0
'Clear screen to blue
Color , 1
Cls
'Create view port
View Print vtop To vbot
'Print the current text
PrintText idx
'Get the current wheel setting
Getmouse mx, my, oldwheel
```

```
55   Do
56       'Get mouse wheel info
57       Getmouse mx, my, mwheel
58       If mwheel <> oldwheel Then
59           'Get delta amount
60           delta = oldwheel - mwheel
61           'Use delta amount to adjust text index
62           idx = idx + delta
63           'Make sure text isn't out of bouns
64           If idx > 19 Then idx = 19
65           If idx < 1 Then idx = 1
66           'Print text
67           PrintText idx
68           'Save the current mouse wheel value
69           oldwheel = mwheel
70       End If
71       Sleep 10
72   Loop Until Inkey<>""
73   'Turn on text cursor
74   Locate ,,1
75
76   End
```

Listing 15.5: view-print.bas

`Analysis:` Line 4 creates a single-dimension array that initialized to some random text. The text will be printed in the view port created later in the program. Line 25 and 26 creates the working variables. Idx is the current line index into the myText array. Mx and my are the mouse column and rows, which are not used in the program, but are needed for the Getmouse function. Oldwheel and mwheel are the last and current wheel position variables. Delta is the difference between the old wheel position and the current wheel position. Delta is used to calculate the offset for the idx variable. Line 26 dimensions two variables, vtop and vbot which are the top and bottom of the view port.

Line 28 through 40 define the PrintText subroutine which will print the array text to the view port. The tidx parameter, is the current value contained within the idx variable, and points to the current line position in the text array. Line 29 creates two variables, i which is used in the following For-Next block and tbot which is used to calculate how many rows of text to print. Line 32 and 33 clear the view port. Line 35 calculates the number of lines to display. Tidx contains the start line in the text array, and tbot contains the end line in the text array. Line 36 checks to make sure that the end line contained within tbot does not point past the end of the array. Line 37 through 39 print the text to the view port. Since the view port is 10 lines, the code prints a 10 line “window” from the array to the view port.

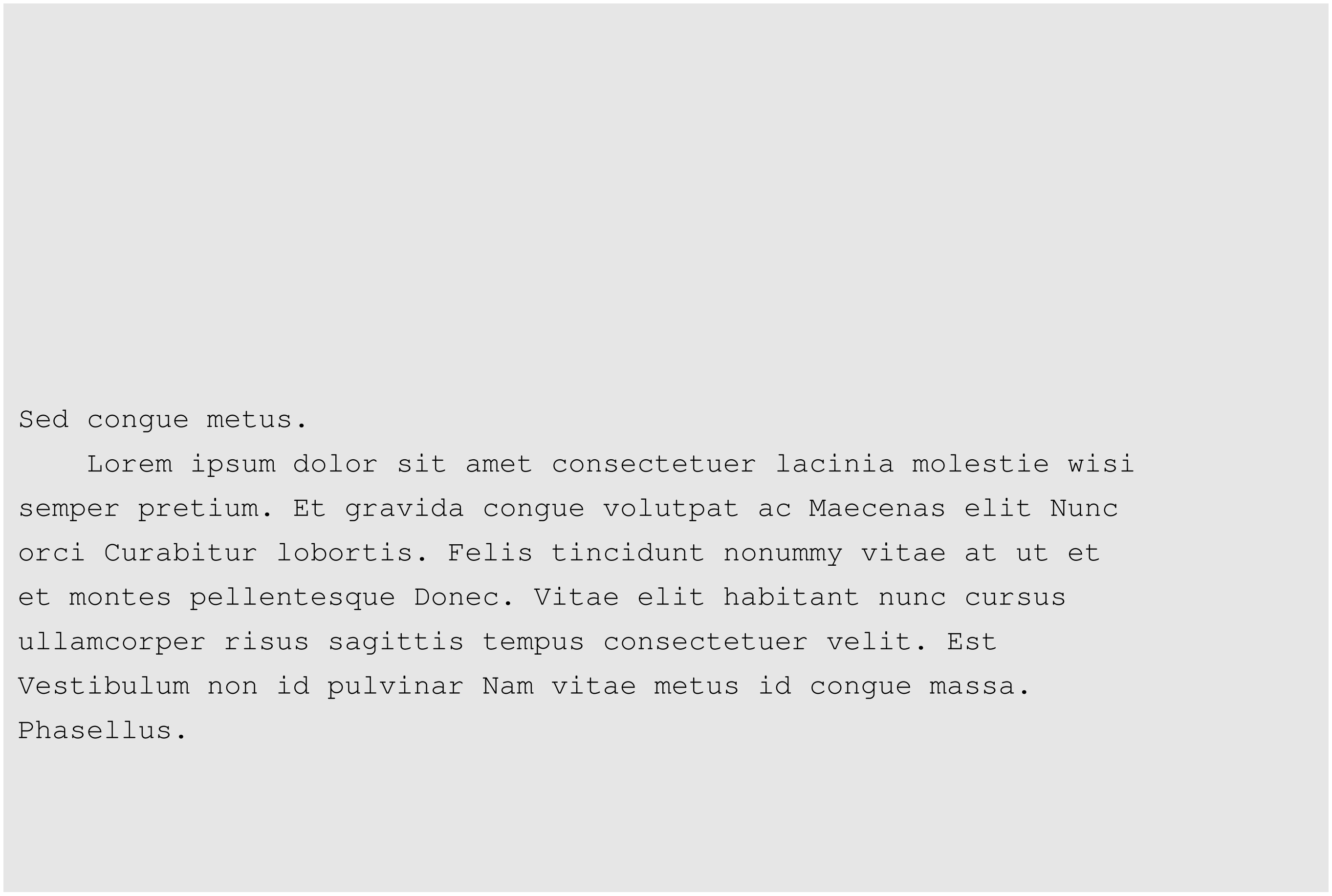
Line 43 sets the width of the console, and lines 45 through 48 clear the main console window to a blue color. At the point the view port has not be created so these commands apply to the console widow as a whole. Line 505 creates a view port that starts at vtop and extends to vbot, inclusive. All text commands issued after the View Print command will go now to the view port. Line 52 prints the initial 10 lines of text. Line 54 gets the initial value of the mouse wheel, for use later to calculate the delta value.

Line 55 through 71 define the mouse input loop. Line 57 gets the current mouse wheel value. Line 58 checks to see if the new value is different than the old value, and if they are different, this means the mouse wheel has been moved. Line 60 calculates the delta, or changed value, between the old mouse wheel setting and the new mouse wheel settings. Using this technique, you can calculate how to scroll the text, because the value will be either positive or negative depending on whether the mouse wheel is moved forward or backward.

Line 62 uses the delta value, to update the idx variable, which is the pointer to the current top line in the text array. If delta is positive, idx will move toward the end of the array, which means that the text will scroll up. If delta is negative, idx will move toward the beginning of the array, so the text will scroll down. Line 64 and 65 make sure that idx does not point past the beginning and end of the array. The current text block is then printed in line 67. Line 69 then saves the current wheel value in the oldwheel variable, to be used in the next delta calculation.

If a key pressed the loop is exited in line 72 and then program is closed with an End statement.

When you run the program you should see output similar to that shown below.



```
Sed congue metus.
    Lorem ipsum dolor sit amet consectetuer lacinia molestie wisi
semper pretium. Et gravida congue volutpat ac Maecenas elit Nunc
orci Curabitur lobortis. Felis tincidunt nonummy vitae at ut et
et montes pellentesque Donec. Vitae elit habitant nunc cursus
ullamcorper risus sagittis tempus consectetuer velit. Est
Vestibulum non id pulvinar Nam vitae metus id congue massa.
Phasellus.
```

Output 15.6: Output of view-print.bas

When you run the program you will see a blue screen with a black view port in the middle with some text. Scroll the mouse wheel, and the text will scroll. Move the mouse wheel forward and the text will scroll down, move the mouse backward and the text will scroll up. The text will stop scrolling when the beginning or the end of the text is reached. As you can see, using the view port command along with the Getmouse function is a simple way to add a great deal of functionality to your text-based programs.

A Look Ahead

Now that you can create variables and gather input, you need to see how to build control structures to act on this data. In the next chapter you will see the various control structures that are available in FreeBasic.

16 Control Structures

Controls structures allow you to change the flow of your program, to execute different parts of your program based on certain criteria and to iterate through code blocks. Control structures fall into two categories; decision-making structures such as the If and Select Case statements, and looping structures such as the For-Next and Do-Loop statements.

The decision-making, or branching structures, allow you define criteria based on data values, and then execute code based on those data values. If a variable is less than 5, you can execute one code block, and if the variable is greater than 5, you can execute a different code block, and if the variable is equal to 5, you can execute yet another code block. Being able to make decisions within your program is crucial for implementing the logic that your program describes.

The looping structures are also necessary elements to any program, since you will often need to process groups of information in a loop, such as reading a disk file or getting input from the user. Data manipulation almost always involves looping, since you will need to iterate through data structures to gather specific information that you will need in your program. FreeBasic has a rich set of both decision-making and looping structures.

A Program is a State Machine

You can think of a program as a state machine, where at any given moment, the program is in a particular state, or condition, which changes over time. This may sound a bit cryptic, so an example may help illustrate what this means. Suppose it is getting close to evening and you realize that you are getting hungry. Call this the hungry state. Your stomach is empty, and your brain is telling you to get some food. You run down to the local hamburger joint, get a nice juicy double cheeseburger and consume it. You are now full, so call this the satisfied state. This is basically how a state machine operates. At one moment, your stomach is empty and the state is hungry and at the next moment, your stomach is full and the state is satisfied. You started in one state (hungry), executed a branching action (drove to the burger joint) and ended in another state (satisfied). Programs do much the same thing.

A variable is like your stomach. If the variable is empty, the program is in the empty state; that is, there is no meaningful data within the variable, so the program cannot do any meaningful work with that data. Programs are designed to do something, so you need to transition from the empty state to the satisfied state, by executing a branching statement (driving down to the burger joint) in order to initialize the variable with data. Once the variable is initialized (you ate the hamburger) the program changes state and now you can do something meaningful, since you now have meaningful data.

Control structures are the transition mechanisms that enable you to change the state of your program. If you are writing a game, you can use a Do-Loop to wait on user input. Once the user presses a key, you can then use an If or Select Case statement block to act on this key press. The program moves from a waiting state, to an action state, and then back to a waiting state, until another key is pressed or the program terminates. To effectively use control structures within your program, you must keep in mind the various states of your program, both before and after state transitions.

The If Statement Block

You can think of the If statement block as a question that requires a True or False answer. The answer determines which section of code your program will execute. Since computers only work with numbers, you frame the question as a conditional equation that will result in either 0 for False or non-zero for True. The following table lists the conditional operators that you can use within an If statement.

Operator	Syntax	Comment
Equal	If var = 5 Then	If operand on both sides of the = are equal, then execute code after Then statement.
Inequality	If var <> 5 Then	If operands on both sides of the <> are not equal, then execute code after Then.
Less Than	If var < 5 Then	If first operand is less than second operand then execute code after Then.
Less Than or Equal	If var <= 5 Then	If first operand is less than or equal to second operand, then execute code after Then.
Greater Than	If var > 5 Then	If first operand is greater than second operand, then execute code after Then.
Greater Than or Equal	If var >= 5 Then	If first operand is greater than or equal to second operand then execute code following Then.

Table 16.1: Logical Operators

You can mix arithmetic, logical and conditional operators, as well as parenthesis, within an If statement. The compiler evaluates the conditional statements from left to right, taking into account the precedence of the operators. For example, all of the following code snippets are legal If statement constructs.

```
If var1 = 5 Then
If (var1 = 5) And (var2 < 3) Then
If var1 + 6 > 10 Then
```

All of these conditional expressions are valid constructs, and if they evaluate to True, or non-zero, then the code following the Then clause will be executed.

Using Bitwise Operators in an If Statement

Remember that the operators And, Or and Not are bitwise operators. That is, they return a value based on the bitwise operation that they perform. You should take care when using bitwise operators within an If statement to make sure that the result will evaluate correctly. Take the second code snippet listed above. If var1 = 5, the compiler will return True, or -1 for the expression. If var2 is less than 3 then the compiler will return True or -1 for this expression. The compiler will then evaluate the And operator with -1 And -1 returning -1. Since -1 is non-zero, or True, the code following the Then will be executed. If either of the statements within the parenthesis evaluate to 0, then And will return 0, which is False, and the code following the Then clause will be skipped. When using bitwise operators you should frame the conditional expressions on either side of the bitwise operator so that they return either True or False. This will give you consistent results in your evaluations.

The Not Problem

The Not bitwise operator can be a problem in an If statement. You may be used to writing `If Not var Then`, with Not performing a logical, rather than a bitwise operation. In FreeBasic Not performs a bitwise operation, not a logical operation. If var were to contain the value of 3, then Not 3 is -4, which will be regarded as a True result and the code following the Then will be executed, which is probably not what you wanted. Instead of writing `If Not var Then`, you should write `If var <> 0 Then`.

The Single-Line If Statement

The basic form of the If statement will execute a single statement if the conditional expression evaluates to True.

```
If <expression> Then <statement>:<statement>...
```

In this form, <expression> can be a single or compound expression and <statement> can be more than one statement separated with : or colons. The following program demonstrates the syntax of the single-line If.

```
Option Explicit

Dim As Integer myInt = 1

If myInt Then Print "This is the first statement." _
               :Print "This is the second statement."

myint = 0
If myInt Then Print "This is the first statement." _
                :Print "This is the second statement."

Sleep
End
```

Listing 16.1: if-single.bas

`Analysis:` Line 3 creates an integer value and initializes the variable to 1. Line 5 uses a single-line If to evaluate the variable. Since myInt is one, which evaluates to True, the compound statement following the Then clause if executed. In line 8, myInt is set to 0. The If statement in line 9 evaluates the variable, which is now 0 or False, so the compound statement following the Then clause is not executed.

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

```
This is the first statement.
This is the second statement.
```

Output 16.1: Output of if_single.bas

You will notice that there is only one set of print lines in the output, since the second If evaluated to False and the print statements were never executed.

The If Code Block

A better way to group multiple statements within an If is to use the If-End If code block. By separating the statements inside a code block, it is much easier to read and to debug if necessary.

```
If <expression> Then
    <statement>
    <statement>
    ...
End If
```

The block If statement operates in the same manner as the single-line If. If <expression> evaluates to True, then the statements between the If and End If are executed.

Nested If Statements

At times it may become necessary to nest If statements in order to better describe the decision making process of the evaluation. While the If statement can handle multiple arguments within an expression, there are times when you may want to incrementally check for certain ranges of values which you can do using a nested If block.

```
If <expression> Then
    <statement>
    ...
    If <expression> Then
        <statement>
        <statement>
        ...
    End If
End If
```

It is important to close each block properly when opened by an If to avoid compiler or logical errors. Compiler errors are fairly easy to fix, while logical errors can be tricky to track down and correct. The best way to make sure you are closing the blocks properly is to indent the nested If statements and then indent the matching End If statements at the same level. In the example above, the indentation tells you at a glance which End If goes with which If.

The Else Statement

When an If expression evaluates to True, the statements following the Then will execute. If the expression evaluates to False, then the statements following the If will execute. However, there are times when you will want to execute a different set of instructions if the If evaluates to False, rather than just continuing on with the program. The Else statement allows you code one or more statements to handle the False case of an If evaluation.

```
If <expression> Then
    <statement>
    <statement>
    ...
Else
    <statement>
    <statement>
    ...
End If
```

The If-Else-End If construct allows you to handle both the True and False cases of an If evaluation. One example of using this format is when you try to open a file. If the Open succeeds, then you can process the file in the statements following the Then clause. If the Open fails, you can display an error message to the user in the Else block, set a flag that indicates the file could not be opened or End the program at this point. How you handle the exception depends on the needs of your program, but the Else statement allows you handle these types of program exceptions.

The ElseIf Statement

The ElseIf statement enables you to create an “If Ladder” construct, where you can test a series of conditions much like the rungs of a ladder. FreeBasic will evaluate the expressions in order, from top to bottom, and then execute the first series of statements that evaluate to True, skipping the rest of the ElseIf tests. Elseif can be used in conjunction with a final Else statement to execute a default set of statements if none of the expressions in the ladder evaluate to True.

```
If <expression> Then
    <statement>
    <statement>
```

```
       ...
ElseIf <expression> Then
       <statement>
       <statement>
       ...
Else
       <statement>
       <statement>
       ...
End If
```

You start the block with a standard If statement, then add any additional ElseIf statements as necessary, and optionally, as in the example, you can add an Else statement as a default handler. The order of an If-ElseIf block can be important. The compiler will evaluate the first If statement, and if the expression is False, it will then drop down to the next ElseIf and evaluate that expression, and continue on down the ladder until an End If or Else is encountered. If your expressions are range values or compound statements then you need to make sure that the expression ranges don't overlap otherwise you may execute the wrong set of Then statements. The following program shows an example of using the ElseIf statement.

```
1     Option Explicit
2
3     Randomize Timer
4
5     Dim As Integer myNumber, i
6
7     For i = 1 to 10
8         myNumber = Int(Rnd * 10) + 1
9         'Build an if ladder
10        If (myNumber > 0) And (myNumber < 4) Then
11            Print "Number is between 1 and 3:";myNumber
12        Elseif (myNumber > 3) And (myNumber < 8) Then
13            Print "Number is between 4 and 7:";myNumber
14        Else
15            Print "Number is greater than 7:";myNumber
16        End If
17    Next
18
19    Sleep
20    End
```

Listing 16.2: elseif.bas

`Analysis:` Line 3 initializes the random number generator with the Timer statement. Line 5 creates two integer variables, myNumber which will hold a random integer between 1 and 10 and i, which will be used in the For-Next statement. Line 7 opens the For-Next block which will execute the code between the For and Next 10 times. Line 8 uses the Rnd function to generate a random number in the rand 1 to 9. Since Rnd returns a double precision number, the Int function is used to convert the value to an integer. Line 10 through 16 define the If ladder. Line 10 checks to see if the random number is between 1 and 3; that is, if the number is 1 it will be greater than 0 and less than 4. Using > and < in this manner defines a range of values to test. Line 12 uses an ElseIf to test for the next range of values between 4 and 7 inclusive. Line 14 defines the default code handler using an Else statement. If the If and ElseIf evaluate to False, then the Else statement will execute. Line 17 closes the For-Next block.

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

```
Number is between 1 and 3: 3
Number is between 4 and 7: 7
Number is between 1 and 3: 1
Number is between 4 and 7: 5
Number is between 1 and 3: 2
Number is between 4 and 7: 7
Number is between 4 and 7: 4
Number is greater than 7: 10
Number is greater than 7: 9
Number is between 4 and 7: 6
```

Output 16.2: Output of elseif.bas

This example program is a case where you must be careful not to overlap the range values, otherwise you will get results you did not expect. Using the > and < operators create a range that is 1 more than the second operand of > and 1 less than the second operand of <. If you were to incorrectly overlap the 4 value for example, then the ElseIf statement would execute for a value of 4, which is not the intention of the program.

The IIF Function

The Iif, or “immediate If” function returns one of two numeric values based on an expression. You can think if the Iif function as an in-line If statement that acts as a function call.

```
Value = Iif(<expression>, numeric_value_if_true, numeric_value_if_false)
```

Iif can be as a standalone function or inside other expressions where you do not want to split the expression to insert an If statement. The numeric values can be literal values, variables or numeric function calls. The limitation of the function is that it will only return a numeric value, not a string value, however you can work around this limitation by using pointers.

Unlike the Visual Basic version of this function, FreeBasic does evaluate both of the return statements.

The following program shows the different ways to use the Iif function.

```
Option Explicit

#define False 0
#define True Not False
#define Null 0

Dim As Integer iresult, a, b, c, d
Dim As Zstring Ptr myZString

Function ResultTrue As Integer
    Print "<In function ResultTrue.>"
    Return True
End Function

Function ResultFalse As Integer
    Print "<In function ResultFalse.>"
    Return False
End Function

Function GetString As Zstring Ptr
    Dim As Zstring Ptr zp

    Print "<In function GetString.>"
    'Need to add 1 for the terminating Null character
    zp = Callocate(Len("Here is a string.") + 1)
    *zp = "Here is a string."
    Return zp
End Function

a = 5
b = 6
c = True
d = False

'Return literal values via the define
iresult = Iif(a > b, True, False)
Print "IIF with values:";iresult

'Return variable values
```

```
40   iresult = Iif(a = b, c, d)
41   Print "IIF with variables:";iresult
42
43   'Return results from numeric function calls
44   iresult = Iif(a < b, ResultTrue, ResultFalse)
45   Print "IIF with functions:";iresult
46
47   'Return a pointer to a string
48   myZString = Iif(a < b, GetString, Null)
49   If myZString <> Null Then
50       Print myZString[0]
51       Deallocate myZString
52   End If
53
54   Sleep
55   End
```

Listing 16.3: iif.bas

Analysis: Lines 3 thorough 5 define the constant values that will be used in the program. False is defined as 0, and True is defined as Not False which will evaluate to -1. Null is defined as 0, which will be used as a check for the pointer variable declared later in the program. Lines 7 and 8 create the working variables for the program. The iresult variable is used as the return value for the Iif function, while the rest of the variables are used as expression and return variables. The pointer variable myZString, is used as the return value from the function GetString.

Lines 10 through 13 and 15 through 18 define two functions that will be used for the True and False return values of the Iif function. Each function prints a string to the screen to indicate that the function was called, and returns either True or False. Lines 20 through 28 define the GetString function that will allocate a memory segment (line 25), initialze the memory segment to a string value (line 26), and return a pointer to this memory segment (line 27). The function also prints a string to the screen in line 23 to indicate that the function was called.

Lines 30 through 33 set the variables to their default values. Line 36 shows how to return a literal value from the Iif function. Since the #define statement will replace the True and False with the defined literal values, this works in the same manner as typing in -1 and 0 into the Iif function. Line 40 shows the Iif function using variables as the return values. Line 44 illustarates using Iif with function calls for the return values. Line 48 illustrates how to return a pointer to a string. Notice that Null is returned in the False part of the Iif function; this is to prevent using a null pointer in the program, since you can check the return value against the Null to be sure that the pointer was properly initialized. Lines Lines 49 through 52 check to see if myZString contains a valid pointer assignment and then prints the string value using the pointer index method in line 50. Line 51 deallocates the memory segment, although in this instance it isn't really necessary, as the memory segment will automatically be deallocated when the program ends. However, it is a good reminder that if you were using this function within a program, such as a loop, you would want to deallocate the memory segment when you finished using it. Good memory management is a good habit that you cultivate.

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

```
IIF with values: 0
IIF with variables: 0
<In function ResultTrue.>
IIF with functions:-1
<In function GetString.>
Here is a string.
```

Output 16.3: Output of iif.bas

As you can see from the example program, the Iif function has a lot of flexibility. Being able to return a function value allows you to extend the functionality to strings and composite types, which you cannot do with the basic format of the function.

The Select Case Statement Block

The Select Case block can be viewed as an optimized if ladder, and works in much the same way. The standard `Select Case` can use any of the standard data types for <expression> and the specialized `Select Case As Const` format is optimized for integer values.

```
Select Case <expression>
        Case <list>
                <statement>
                <statement>
                ...
        Case Else
                <statement>
                <statement>
                ...
End Select
```

This code snippet shows the syntax of the standard select case. Expression is usually a variable which can be of any of the standard data types, or individual elements of a Type or array. The <list> clause of the Case statement can be any of the following formats.

- Case <value>: Value is one of the supported data types or an enumeration.
- Case <value> To <value>: Specifies a range of values.
- Case Is <operator> <value>: Operator is any of the logical operators listed in Table 17.1.
- Case <value>, <value>, ...: List of values separated with commas.

- Case <variable>: A variable that contains a value.

```
Select Case As Const <integer_expression>
       Case <list>
              <statement>
              <statement>
              ...
       Case Else
              <statement>
              <statement>
              ...
End Select
```

The Select Case As Const is a faster version of the Select statement designed to work with integer expressions in the range of 0 to 4097. The <list> statement formats for the Select Case As Const are limited to values or enumerations of values. That is, the operator expressions are not allowed within a Case As Const.

When a Case block is executed, the statements following the Case keyword up to the next Case keyword (o End Select) will be executed. Only one block of statements within a Case will execute at any one time. If a Case Else is present, then the statements within the Else block will execute if no Case matches the <expression> portion of the Select statement. The following program illustrates using the Select Case statement block.

```
Option Explicit

'Ascii code of key press
Dim As Integer keycode

'Loop until esc key is pressed
Do
    keycode = Asc(Inkey)
    Select Case As Const keycode
        Case 48 To 57
            Print "You pressed a number key."
        Case 65 To 90
            Print "You pressed an upper case letter key."
        Case 97 To 122
            Print "You pressed a lower case key."
        End Select
        Sleep 1
Loop Until keycode = 27 '27 is the ascii code for Escape

End
```

Listing 16.4: selectcase.bas

Analysis: Line 4 declares an integer value that will be used to capture the ascii codes in the Do-Loop block. Line 7 opens the Do-Loop block. Line 8 uses the Inkey and Asc functions to return the ascii code of a key press. Lines 9 through 16 define the Select Case As Const block. Line 9 uses the variable keycode as the expression for the select case. Line 10 checks to see if the keycode variable is between 48 and 57 inclusive, and if True will execute the code in line 11. Line 12 checks to see if the ascii code falls in the upper case letter range and if True, will execute line 13. Line 14 checks the lower case letter range and executes line 15 if True. Line 16 closes the Select Case block.

Sleep is used in line 17 to allow other process to run while this program is running. Line 18 checks to see if the keycode variable contains 27, the ascii code for the Escape key, and exits the loop if True. The program is then closed in the usual way.

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

```
You pressed an upper case letter key.
You pressed an upper case letter key.
You pressed a lower case key.
You pressed a lower case key.
You pressed a number key.
You pressed a number key.
```

Listing 16.5: Output of selectcase.bas

You wil notice that when the program runs, only one Case statement is executed at any one time. If you have used C, then the Select Case is like the Switch statement, but with implicit break statements within each Case block. Since only one Case block will execute, you cannot have fall-through situations like you have in C.

17 Appendix A: GNU Free Documentation License

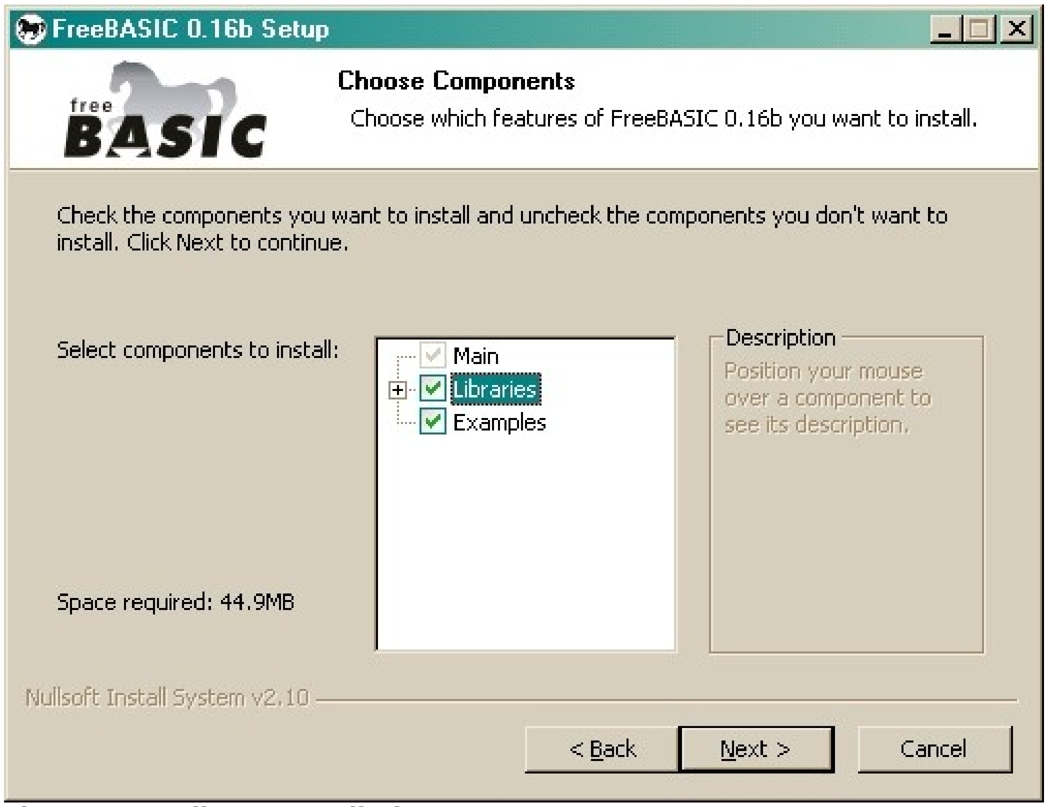
18 Appendix B: Setting Up FreeBasic Under Microsoft Windows

The first step in programming in FreeBasic is to download the compiler package and install it. You can skip this section if you already have FreeBasic installed on your system.

You can find the current version at http://www.freebasic.net/. At the time of this writing, the current stable version of FreeBasic is .17b. The instructions presented here are for the current version. Later versions may have a different installation method so be sure to read the included documentation before installing the package. Click on the download link on the main page of the FreeBasic website and select the Windows stable version. You will be presented with a list of download mirrors. Select the site closest to you (which will download faster) and save the file to a known location of your hard drive.

The filename should be FreeBASIC-v0.17b-win32.exe, or something similar. Double click this file to run the setup program. Click Next on the first screen, read through the license on the second screen and click the Accept button. FreeBasic is licensed under version 2 of the Gnu General Public License[5]. The third screen allows you select the files that will be installed. For the purposes of the book, be sure to install all libraries by clicking on the Libraries check box until you see a green check mark, as illustrated in Figure 1.1.

[5] See the Free Software Foundation's GNU website http://www.gnu.org/copyleft/gpl.html for details of the GPL license.


FreeBASIC 0.16b Setup

Choose Components

Choose which features of FreeBASIC 0.16b you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [✓] Main
- [✓] Libraries
- [✓] Examples

Description

Position your mouse over a component to see its description.

Space required: 44.9MB

Nullsoft Install System v2.10

Figure 18.1: Library Installation

You also want to make sure the examples are installed as well as the libraries, since there is a wealth of information to be gleaned from the provided examples. Click Next to select the location where you want to install the package and then click Next again to select where you want to place the shortcuts on your system menu and then click Install.

The setup program will install the files. This may take some time depending on the speed of your computer. Once all the files have been installed click finish. If the Run FreeBasic checkbox is checked, then a command window will open in the FreeBasic folder.

Installing FBIde

Unlike QuickBasic, FreeBasic does not have a built-in editor, so a third party editor must be used to create programs. Any text-based editor can be used and there are many free programmers' editors available on the Internet. FBIde is an editor specifically designed to be used with FreeBasic and is the editor used for all the programs in this book. If you do not have FBIde already installed, then you must download the editor from the FBIde website located at http://www.fbide.freebasic.net.

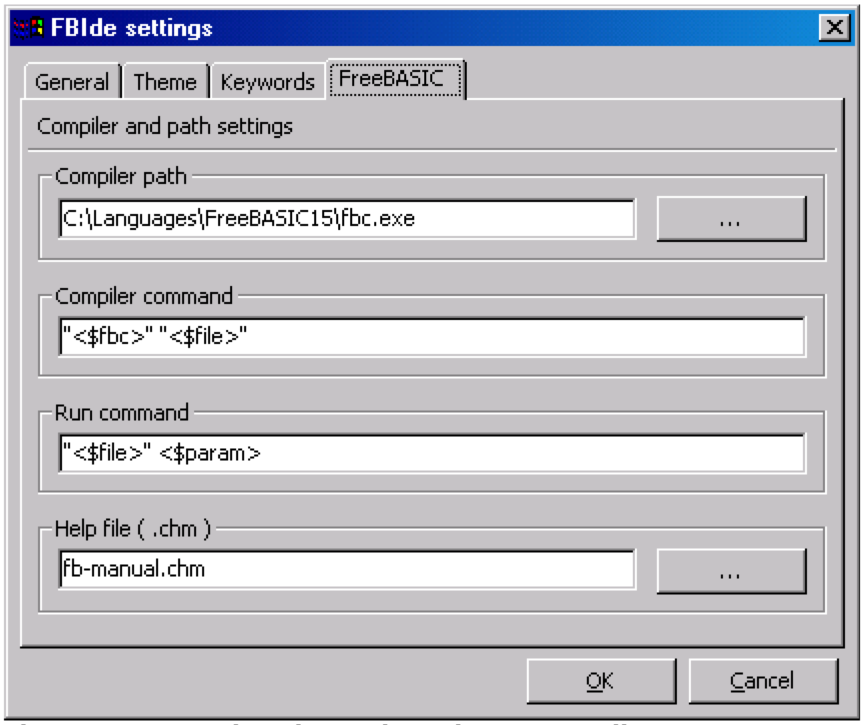
Click on the download link and select the FBIde installer download link. Select the download mirror nearest to your location, for a faster download, and save the file to a known location on your hard drive. At the time of this writing, the current version of FBIde is 0.4.6 and the filename should FBIde_0.4.6.exe or something similar.

Double click the file to install the editor. Click Next on the first screen and make sure the "I accept the agreement" radio button is checked and click Next. Select the folder to install the editor. To make things easier, install FBIde in the folder where you installed FreeBasic. Click the Next button. Select the menu shortcut location, and again you should group the FBIde shortcut with the FreeBasic shortcut created earlier and click Next. The next screen allows you to install desktop shortcuts and associate *.bas and *.bi files with FBIde. If FreeBasic is the only basic compiler you are using, then it is convenient to associate these files with FBIde. If FreeBasic isn't the only basic compiler you have, then you might not want to associate the files. Click the Next button and the final screen will be displayed. Click the install button.

Once the files have been installed click the Finish button. If the Launch Application check box is checked, FBIde will be started. The ide executable is called fbide.exe and should be in the main install folder.

Setting Up FBIde

In order to create FreeBasic programs with FBIde, the ide will need to know the location of the compiler. Start FBIde and select View->Settings from the main menu. Select the FreeBasic tab and make sure the Compiler Path textbox is pointing to the FreeBasic compiler, fbc.exe as illustrated in Figure 1.2.



Compiler and path settings

Compiler path: C:\Languages\FreeBASIC15\fbc.exe

Compiler command: "<$fbc>" "<$file>"

Run command: "<$file>" <$param>

Help file (.chm): fb-manual.chm

Figure 18.2: Setting the path to the FB compiler

Your setup will be different depending on where you installed FreeBasic, but should look similar to the above example. Once the path is setup, you are ready to compile your first program.

Compiling Your First Program

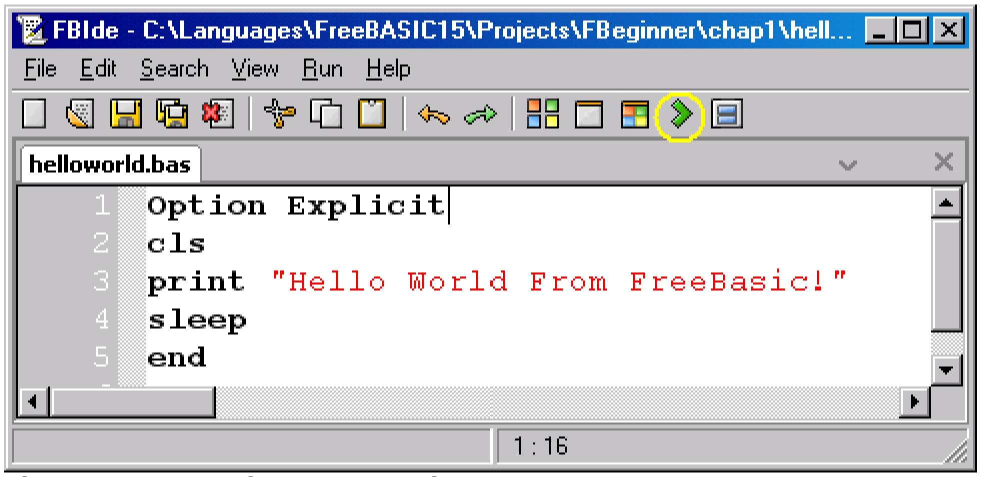
This is a good time to test the installation to make sure everything is set up correctly. Start FBIde, if it isn't already running, enter the following code into the editor and save the code as helloworld.bas.

```
1   Option Explicit
2   Cls
3   Print "Hello World From FreeBasic!"
4   Sleep
5   End
```

Listing 18.1: helloworld.bas

`Analysis:` Line 1 is a compiler directive that tells the compiler each variable should be declared with the Dim statement. This program doesn't contain any variables, but it is a good habit to always use Option Explicit in your programs. The Cls statement in line 2 clears the console screen. The Print statement in line will print the string "Hello World From FreeBasic" at the current cursor position. The Sleep statement in line 4 pauses the program and waits for a key press. The End statement in line 5 ends the program.

FreeBasic is a true compiler, which means that the code must be converted to an executable before you can run the program. Using FBIde however, the process is quite transparent. After typing in the above code click on the green arrow on the toolbar, circled in yellow in Figure 1.3.


FBIde - C:\Languages\FreeBASIC15\Projects\FBeginner\chap1\hell...
File Edit Search View Run Help
helloworld.bas
1 : 16

```
Option Explicit
cls
print "Hello World From FreeBasic!"
sleep
end
```

Figure 18.3: Running a FreeBasic Program

If there are no errors in the program, clicking this button will compile and run the program resulting in the output. This is an example of a console program which uses a text mode window to display content. Most of the programs in this book are console mode programs, with the exception of the graphics programs.

FreeBasic does not have a built-in GUI interface. It relies on third-party libraries such as the Windows SDK, GTK or wxWidgets. This allows the programmer to pick the GUI interface that best meets the needs of the program, and since many of the GUI libraries are cross-platform, will allow GUI programs to run on several operating systems.

```
Hello World from FreeBasic!
```

Output 18.1: helloworld.bas

Press any key to exit the program. If you see the above screen, everything is set up correctly and you are ready to begin writing FreeBasic programs. If you get an error message, make sure that you have typed in the code exactly as listed, and that you have entered the correct path to the compiler in the compiler settings.

Additional Resources

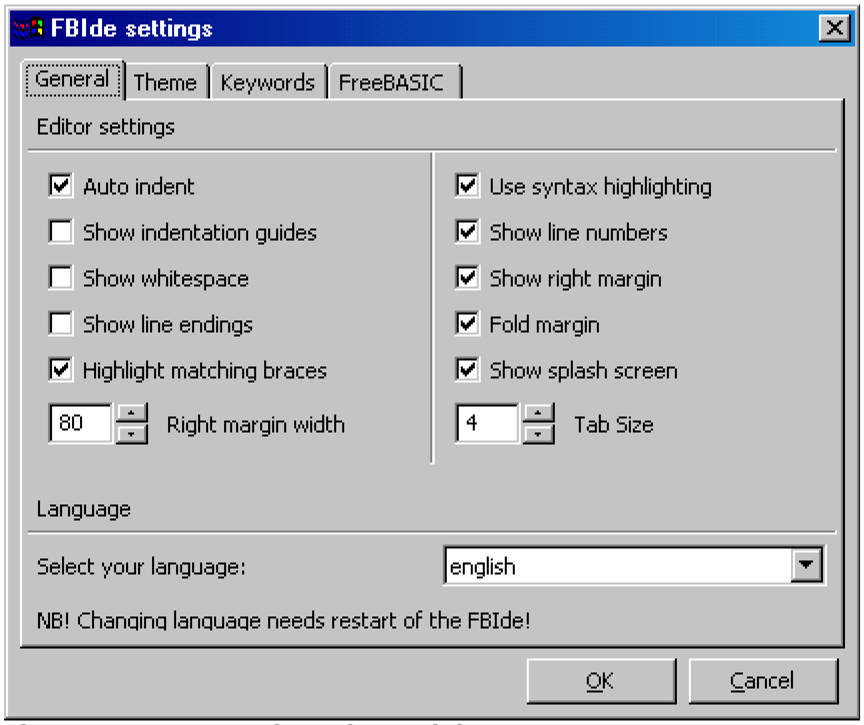
In addition to this book, there are several resources on the Internet that you can use for any questions you may have on installing FreeBasic or on writing programs. You can see the current list of resources at the support page of the FreeBasic website, http://www.freebasic.net.. The FreeBasic community is a generous bunch and are willing to help newcomers over the first hurtles of programming.

Introduction to FBIde

FBIde is an Open Source ide for FreeBasic written by Albert Varaksin. At the time of this writing, FBIde is at version 0.4.6. You should check the FBIde website, http://fbide.freebasic.net/, to see if there are newer versions of the software. This introduction is based on 0.4.6; so later versions may be different.

General Settings

The first task is to set up the FBIde environment. Select View -> Settings from the main menu. You should see the General Settings tab displayed as in Figure 2.1.



Editor settings

- ✓ Auto indent
- ☐ Show indentation guides
- ☐ Show whitespace
- ☐ Show line endings
- ✓ Highlight matching braces
- 80 Right margin width
- ✓ Use syntax highlighting
- ✓ Show line numbers
- ✓ Show right margin
- ✓ Fold margin
- ✓ Show splash screen
- 4 Tab Size

Language

Select your language: english

NB! Changing language needs restart of the FBIde!

Figure 18.4: General Settings Dialog

Make sure that your settings are the same as Figure 2.1 for the purposes of this chapter and click the OK button. You can always change them later as you develop your method of working.

The best way to understand how FBIde works is to look at an example program. Start FBIde and type in the following program. Do not type in the line numbers as they are only provided as reference. Don't worry about the details of the program. This is just an example to illustrate some of the features of FBIde.

```
1    Option Explicit
2    Declare Function GetHello() as String
3    Dim HelloString as String
4    'Clear the screen
5    Cls
```

```
6   'Get the hello string
7   HelloString = GetHello
8   'Print to console window
9   Print HelloString
10  'Wait for keypress
11  Sleep
12  'End program
13  End
14  'GetHello function
15  Function GetHello() as String
16      Return "Hello World from FreeBasic!"
17  End Function
```

Listing 18.2: helloworld2.bas

`Analysis:` Once again the program is started with Option Explicit in line 1. The Declare statement in line 2 tells the compiler that a function GetHello is somewhere in the program. The declaration statement allows you to reference a function before the compiler has compiled the code for the function. In line 3 a string variable has been declared using the Dim statement. When the compiler sees the Dim statement, it will allocate memory based on the data type. Line 4 is an example of a comment. A comment starts with the ' (single quote) character and continues to the end of the line. The Cls statement in line 5 will clear the console screen. In line 7 the string variable previously declared is set the value of the function, GetHello which returns a string value. In line 9 the contents of the string variable are printed to the string. Line 11 through 13 pause the program and wait for a key press.

The function GetHello that was declared in line 2 is defined in lines 15 through 17. The keyword Function indicates that this procedure will return a value, using the Return statement in line 16. Each function must be closed with the End Function keywords shown in line 17.

You should frequently save your work while working on your programs. A lot of unexpected things can happen, and if for some reason the computer shuts down or you lose power, you can easily lose hours of work. Table 2.1 shows the File functions in FBIde. The shortcut keys show the key combination required to invoke the functions. For example, to invoke Undo you would hold down the Ctrl key while pressing the Z key. All the short-cut keys work in the same fashion. Learning the short-cut keys will save valuable time during the editing process.

Function	Meaning	Menu	Short-cut	Toolbar Icon
New File	Creates a new file and opens a blank tab	File -> New	Ctrl-N	🗋
Open File	Displays open file dialog and loads new file into a new tab	File -> Open	Ctrl-O	📂

Function	Meaning	Menu	Short-cut	Toolbar Icon
Save File	Saves current file to disk. If file is new, the save dialog will be displayed	File -> Save	Ctrl-S	💾
Save File As	Save current file with a new file name	File -> Save As	Ctrl-Shift-S	None
Save All	Saves all open files	File -> Save All	None	💾
Load Session	Loads all files saved as a session	File -> Load Session	None	None
Save Session	Saves all open files as a session (similar to a project)	File -> Save Session	None	None
Close	Closes the current open file	File -> Close	Ctrl-F4	✖
Close All	Closes all open files	File -> Close All	None	None
New Window	Opens a new instance of FBIde	File -> New Window	Shift-Ctrl-N	None
Quit	Closes FBIde	File -> Quit	Ctrl-Q	None

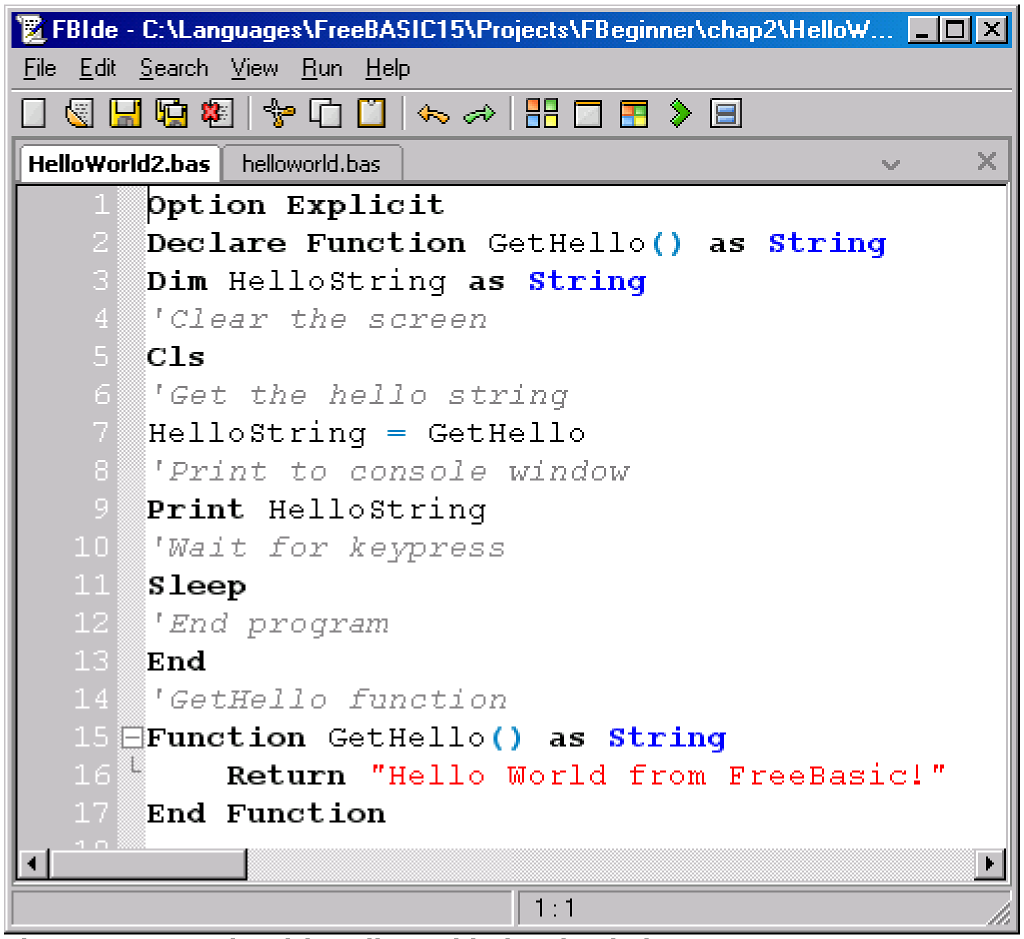
Table 18.1: File Functions

Most Recent Used Files List

The list of files under the Quit menu option is the MRU or most recently used file list. Selecting a file from the list will load the file into the editor.

Syntax Highlighting

After typing in the program, FBIde should look similar to Figure 2.2.


FBIde - C:\Languages\FreeBASIC15\Projects\FBeginner\chap2\HelloW...
File Edit Search View Run Help
HelloWorld2.bas
helloworld.bas
1 : 1

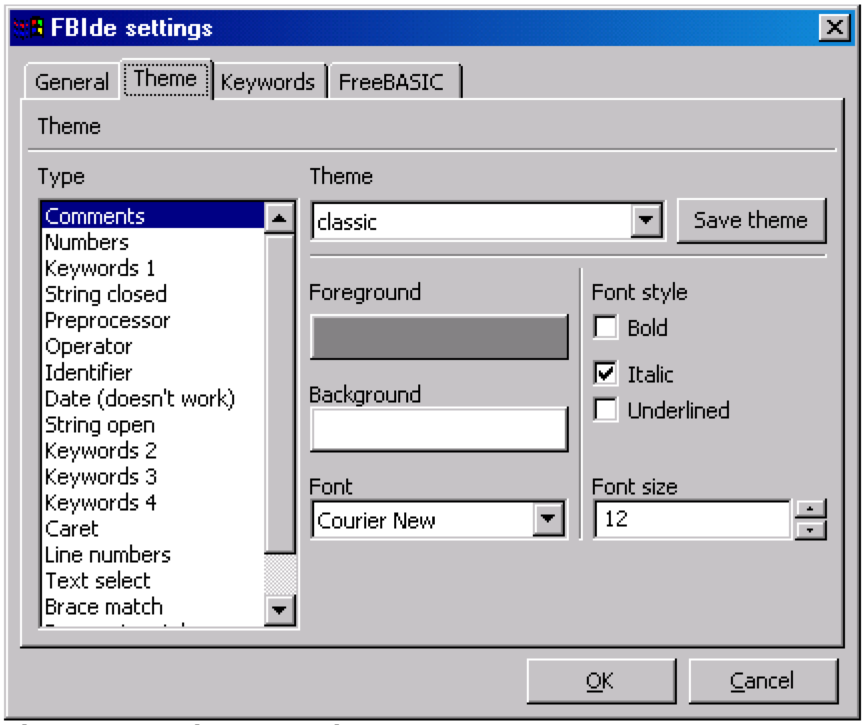
```
Option Explicit
Declare Function GetHello() as String
Dim HelloString as String
'Clear the screen
Cls
'Get the hello string
HelloString = GetHello
'Print to console window
Print HelloString
'Wait for keypress
Sleep
'End program
End
'GetHello function
Function GetHello() as String
      Return "Hello World from FreeBasic!"
End Function
```

Figure 18.5: FBIde with HelloWorld2.bas loaded

The first thing you will notice is that the keywords are highlighted in different colors. This is a visual aid to help indicate the correct spelling of the keywords, data types as well as to indicate data items. You can change the color scheme by selecting View -> Settings from the main menu and selecting the Theme tab.

Themes

There are several predefined themes that you can select by clicking on the Theme drop-down box as shown in Figure 2.3. You can also create your own theme by selecting Create New Theme from Theme drop-down, selecting the language elements from the Type list box, selecting the desired Foreground and Background colors for the selected element, as well as the font characteristics for the language element. Click on the Save Theme button to save the theme for later use.



FBIde settings

Theme

Type

- Comments
- Numbers
- Keywords 1
- String closed
- Preprocessor
- Operator
- Identifier
- Date (doesn't work)
- String open
- Keywords 2
- Keywords 3
- Keywords 4
- Caret
- Line numbers
- Text select
- Brace match

Theme: classic

Foreground

Background

Font: Courier New

Font style

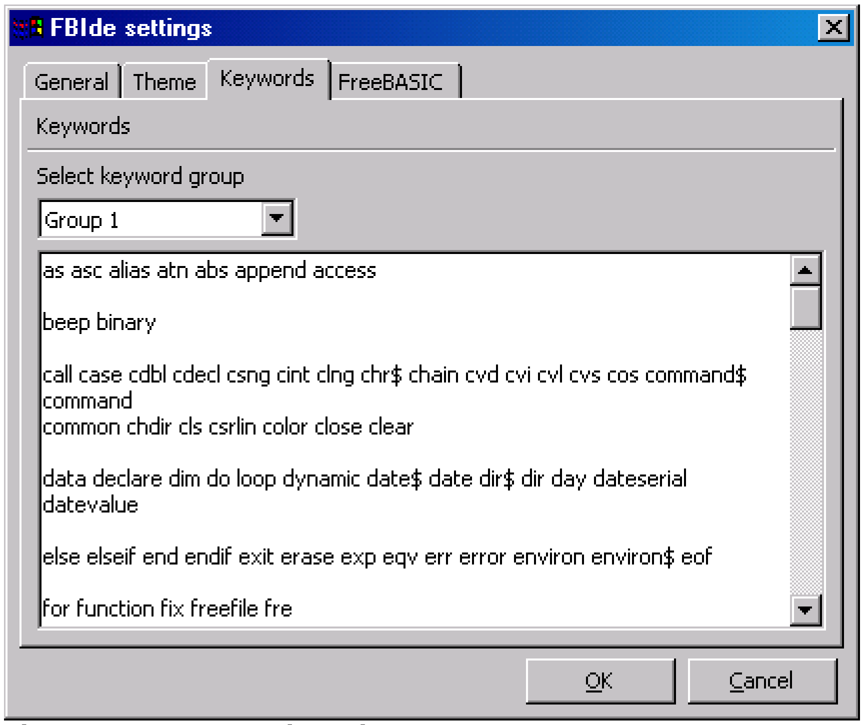
- [ ] Bold
- [✓] Italic
- [ ] Underlined

Font size: 12

Figure 18.6: Theme Settings

Keywords

In order to colorize the keywords, FBIde contains keyword group lists under the Keywords tab of the settings dialog as shown in Figure 2.4.



as asc alias atn abs append access

beep binary

call case cdbl cdecl csng cint clng chr$ chain cvd cvi cvl cvs cos command$ command
common chdir cls csrlin color close clear

data declare dim do loop dynamic date$ date dir$ dir day dateserial datevalue

else elseif end endif exit erase exp eqv err error environ environ$ eof

for function fix freefile fre

Figure 18.7: Keyword Settings

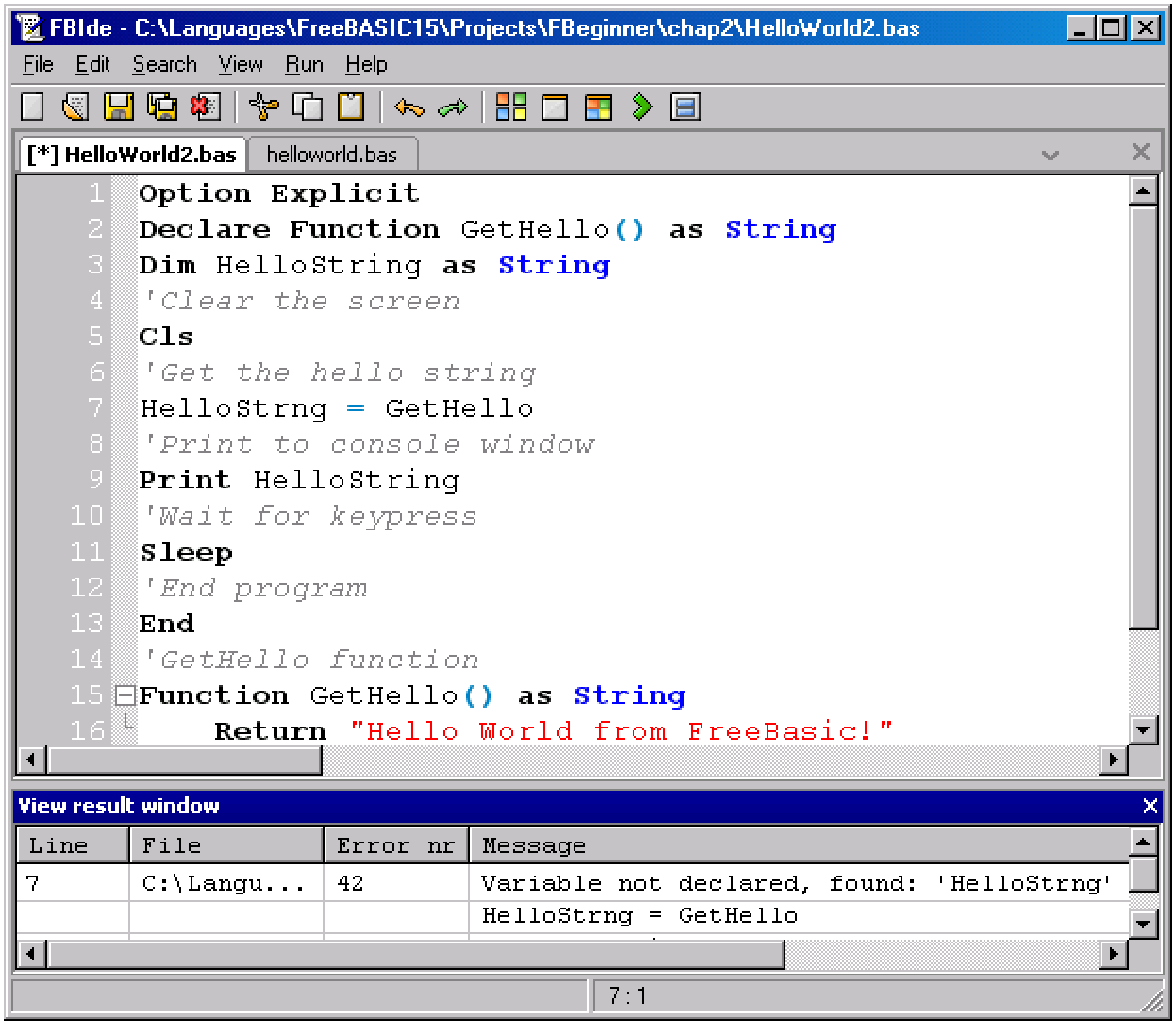
FreeBasic is in continual development, and a new version of the compiler may come out ahead of the FBIde updates. If new keywords are added to the language, you can add them here so that they will be colorized in the editor.

Tabbed Interface

FBIde uses a tabbed interface so you can load more than one file at a time. For large projects, it is a good idea to break up your program into different modules, and each loaded module will have a tab and workspace of its own. The current active file will be the highlighted tab.

Line Numbers

The numbers in the left margin are line numbers and are provided for reference. If there is an error in your program and it won't compile correctly, you will see an error message and line number in the result window. FBIde will also position the cursor on the offending line. Figure 2.5 shows the result window with an error displayed.


FBIde - C:\Languages\FreeBASIC15\Projects\FBeginner\chap2\HelloWorld2.bas
File Edit Search View Run Help
[*] HelloWorld2.bas
helloworld.bas
View result window
7 : 1

```
 1  Option Explicit
 2  Declare Function GetHello() as String
 3  Dim HelloString as String
 4  'Clear the screen
 5  Cls
 6  'Get the hello string
 7  HelloStrng = GetHello
 8  'Print to console window
 9  Print HelloString
10  'Wait for keypress
11  Sleep
12  'End program
13  End
14  'GetHello function
15  Function GetHello() as String
16      Return "Hello World from FreeBasic!"
```

```
Line  File           Error nr  Message
7     C:\Langu...    42        Variable not declared, found: 'HelloStrng'
                               HelloStrng = GetHello
```

Figure 18.8: Result window showing an error

Results Window

The result window indicates line number in the Line column, the current file in the File column, the error number returned by the compiler in the Error Nr column, and the corresponding error message in the Message column, along with the offending piece of code. You will notice that the cursor has been placed on the correct line in the editor.

If you look at the bottom right panel in Figure 2.5 you will see a 7: 1. This indicates that the cursor is currently on line 7 and column 1.

Subroutine and Function Folding

If you look at line 15 in Figure 2.5 you will see a small white box next to the line number. This is the subroutine and function folding handle. Click on the box and the code for the subroutine will collapse and clicking again on the box will display the code. If your program has a number of subroutines or functions, this is a handy way to keep the workspace a bit less cluttered.

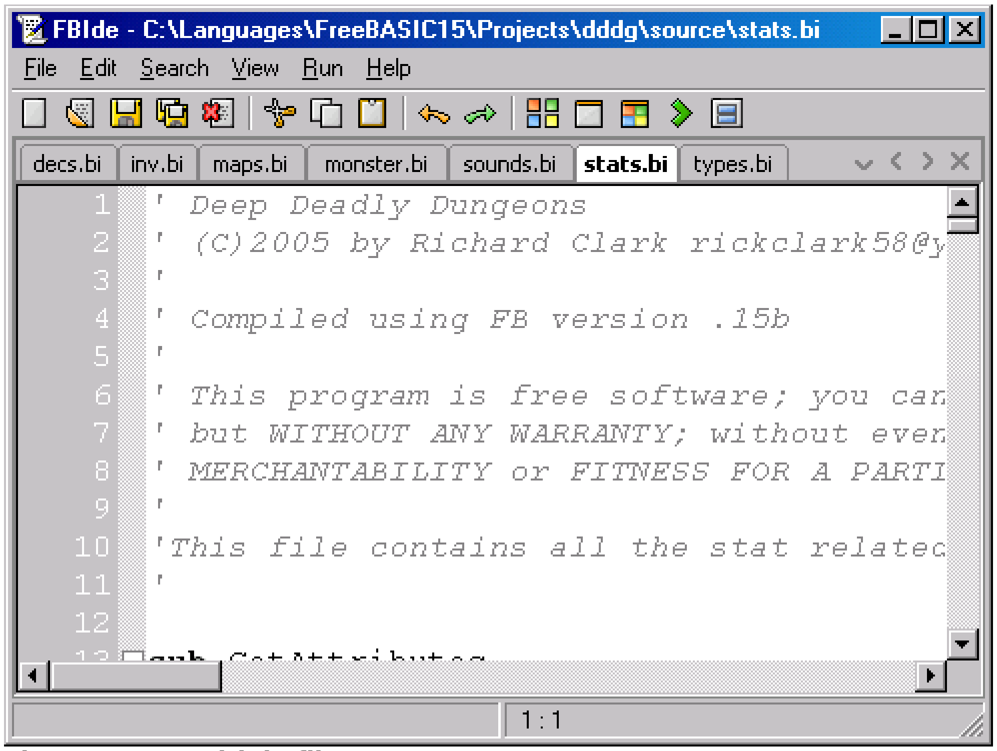
Folding is based on the indentation of your program and not the keywords. If you do not indent your program you will not see any folding handles.

Save File Indicator

Notice in Figure 2.5 that there is an asterisk next to the filename in the highlighted tab. This indicates that the file has not yet been saved.

Quit Tab Select and Close Tab

If you have a lot of files open, you may not be able to fit all the tabs on the screen, as illustrated in Figure 2.6.


FBIde - C:\Languages\FreeBASIC15\Projects\dddg\source\stats.bi
File Edit Search View Run Help
decs.bi
inv.bi
maps.bi
monster.bi
sounds.bi
stats.bi
types.bi
1 : 1

```
' Deep Deadly Dungeons
' (C)2005 by Richard Clark rickclark58@y
'
' Compiled using FB version .15b
'
' This program is free software; you can
' but WITHOUT ANY WARRANTY; without even
' MERCHANTABILITY or FITNESS FOR A PARTI
'
'This file contains all the stat relatec
'

sub GetAttributes
```

Figure 18.9: Multiple files open

The small downward pointing arrow to the right of the tab list is the quick tab select. Clicking on this arrow will display a list of loaded files. Selecting a file from the list will make that tab active. Next to the down arrow are left and right arrows that will scroll left and right through the tab list. The X will close the current file in the selected tab.

Editing and Format Functions

The editor supports all the standard editing and search functions under the Edit and Search menu items. Table 2.2 shows all the editing and search functions.

Function	Meaning	Menu	Short-cut	Toolbar Icon
Undo	Removes last editing change	Edit -> Undo	Ctrl-Z	↶
Redo	Restores last editing change	Edit -> Redo	Ctrl-Shift-Z	↷

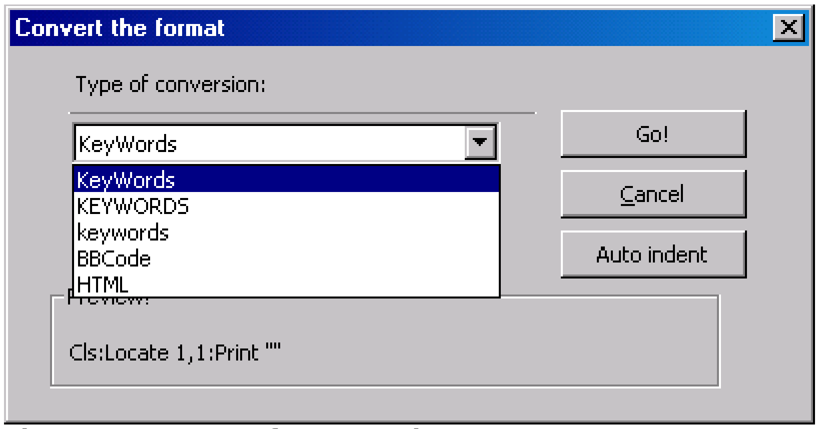
Function	Meaning	Menu	Short-cut	Toolbar Icon
Cut	Cuts selected (highlighted) text to clipboard	Edit -> Cut	Ctrl-X	✂
Copy	Copies selected text to clipboard	Edit -> Copy	Ctrl-C	⧉
Paste	Pastes text into editor from clipboard	Edit -> Paste	Ctrl-V	📋
Select All	Selects all text in editor	Edit -> Select All	Ctrl-A	None
Select Line	Selects all text on current line	Edit -> Select Line	Ctrl-L	None
Indent Increase	Indents all selected lines	Edit -> Indent Increase	Tab	None
Indent Reduce	Removes last indent for all selected lines	Edit -> Indent Reduce	Shift-Tab	None
Comment Block	Adds comment character (‘) to selected lines	Edit -> Comment Block	Ctrl-M	None
Uncomment Block	Removes comment character from selected lines	Edit -> Uncomment Block	Ctrl-Shift-M	None
Find	Displays the find dialog	Search -> Find	Ctrl-F	None
Find Next	Continues last find command	Search –> Find Next	F3	None
Replace	Displays the replace dialog	Search -> Replace	Ctrl-R	None
Goto Line	Displays the goto line dialog	Search -> Goto Line	Ctrl-G	None

Table 18.2: Editing Functions

Block Comment-Uncomment

One nice feature of the ide the block comment, uncomment that you will find under the Edit menu. Highlight one or more lines of code, select Edit -> Comment Block to convert the code to comments. This will add the comment marker ‘ at the front of the code. Select Edit -> Uncomment Block to remove the comments. This is very handy if you want to add notes to the program, or comment out lines of code.

FBIde also has a built-in code formatter that you can find under View -> Format on the main menu.



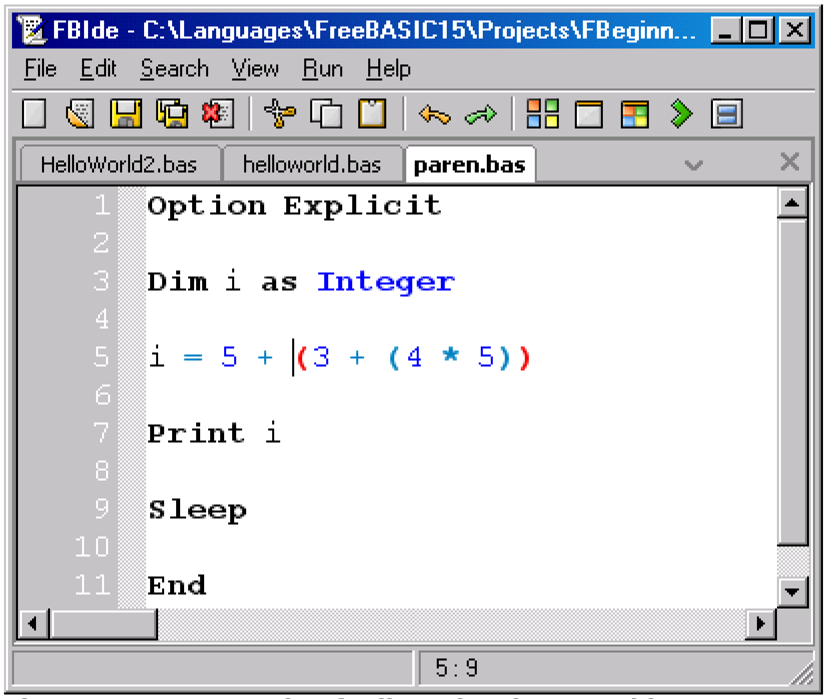
```
Cls:Locate 1,1:Print ""
```

Figure 18.10: Auto-format options

As you can see there are a number of format options, including HTML and BBCode that is used in many popular forums.

Bracket Matching

Another nice feature of FBIde is the Highlight Matching Braces setting in the Settings dialog. Move the cursor to the left of a parenthesis, and the matching parenthesis will be highlighted in the theme color, as illustrated in Figure 2.8.



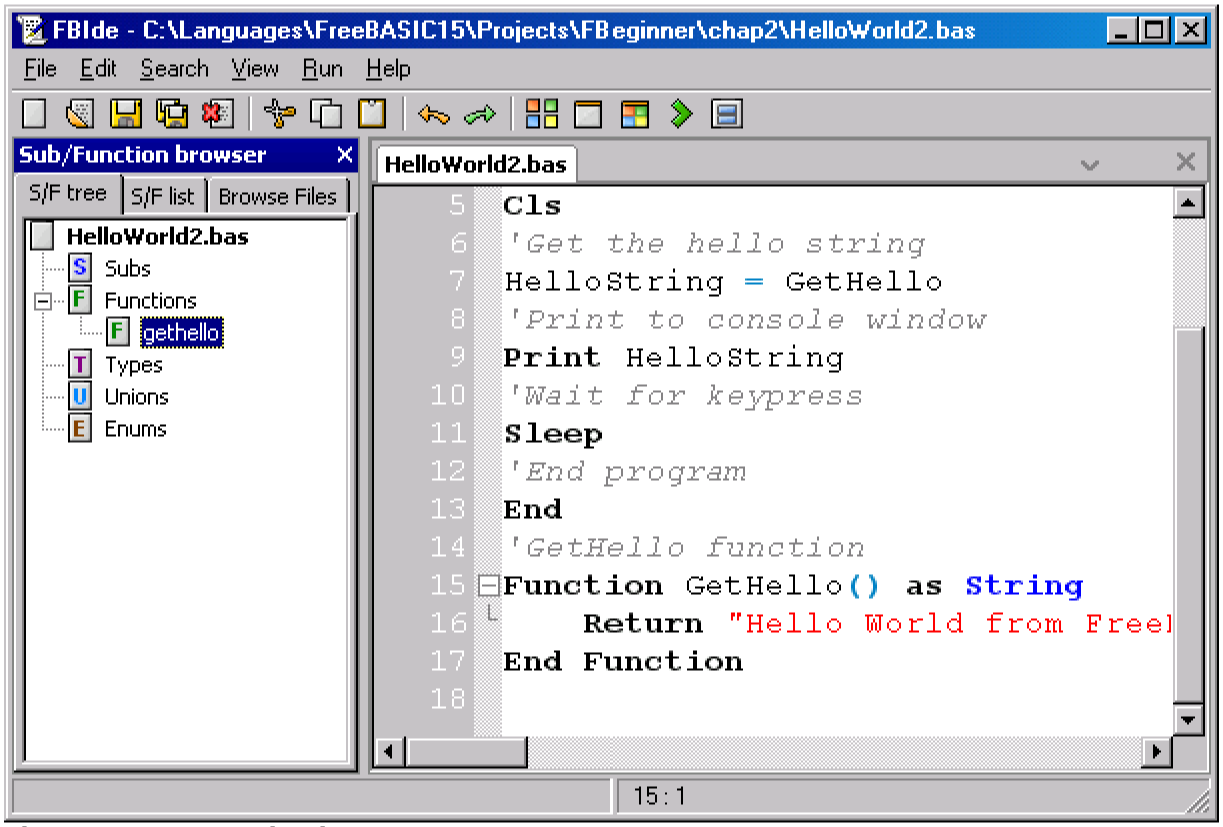
```
Option Explicit

Dim i as Integer

i = 5 + (3 + (4 * 5))

Print i

Sleep

End
```

Figure 18.11: Example of editor showing matching parenthesis

This is very useful when working with complex expressions to make sure that all the parentheses are balanced correctly.

Subroutine and Function Browser

In large programs finding the correct subroutine or function can sometimes be difficult, but FBIde has a function browser, which simplifies the task. The function browser is shown in Figure 2.9.



```
Cls
'Get the hello string
HelloString = GetHello
'Print to console window
Print HelloString
'Wait for keypress
Sleep
'End program
End
'GetHello function
Function GetHello() as String
    Return "Hello World from Free
End Function
```

Figure 18.12: Function browser

Press F2 to show or hide the browser. The browse tree is shown in Figure 2.8. Selecting the S/F List tab, will display the function and subroutine names in a list format. Double-clicking on a function name in the list will display the function in the editor. The Browse Files tab displays an Explorer style file list to browse files on your hard drive.

As you can see in the Tree list, not only does the browser display subroutines and functions, it will also list types, unions and enumerations, which will be covered later in the book.

Running Programs and Creating Executables

The Run menu item has a number of options available for running and creating executables. Table 2.3 explains the meaning of the various menu items.

Function	Meaning	Menu	Short-cut	Toolbar Icon
Compile	Creates an executable file from the source.	Run -> Compile	Ctrl-F9	▦

Function	Meaning	Menu	Short-cut	Toolbar Icon
Compile and Run	Creates an executable from source and then runs the executable.	Run -> Compile and Run	F9	▦
Run	Runs a previously created executable.	Run -> Run	Shift-Ctrl-F9	□
Quick Run	Creates a temporary executable and then runs that executable. The temporary executable is deleted after the program exits.	Run -> Quick Run	F5	▶
CMD Prompt	Displays a console window	Run -> CMD Prompt	F8	None
Parameters	Displays parameter dialog	Run -> Parameters…	None	None
Show Exit Code	Displays program’s exit code	Run -> Show Exit Code	None	None
Active Paths	Sets working folder current during compilation and execution	Run -> Active Paths This is a checked menu item. When checked it is active, when cleared it is not active.	None	None
Open/Close Output Window	Displays or closes the results window	None	None	▤

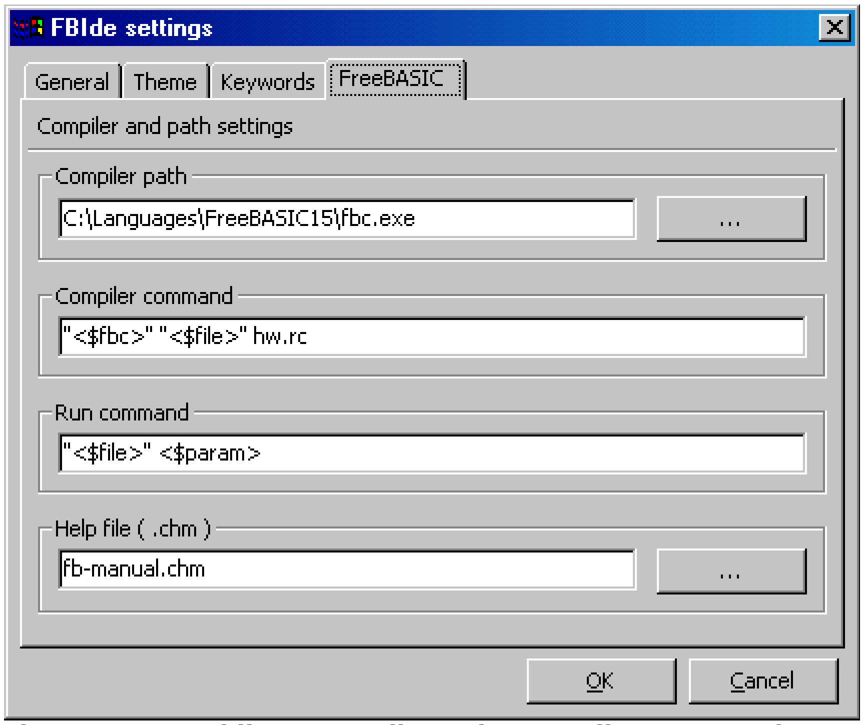
Table 18.3: Run Menu Items

Adding an Icon to Your Program

When you create an executable you will notice that the EXE file has the rather bland, standard windows executable icon. You can add a custom icon to your application by using a resource file. Create a new file in FBIde by selecting File -> New or clicking the New icon on the toolbar. Enter the following line of text into the editor window.

```
FB_PROGRAM_ICON ICON hw.ico
```

Save the file as hw.rc in the same folder as HelloWorld2.bas and close the file. You will need an icon file in the same folder as the rc file with the file name hw.ico. Load HelloWorld2.bas if it isn’t already loaded. Select View -> Settings from the main menu and select the FreeBasic tab on the settings dialog.



Compiler and path settings

Compiler path: C:\Languages\FreeBASIC15\fbc.exe

Compiler command: "<$fbc>" "<$file>" hw.rc

Run command: "<$file>" <$param>

Help file (.chm): fb-manual.chm

Figure 18.13: Adding an rc File to the Compiler Commands

In the Compiler Command text field add hw.rc after the "<$fbc>" "<$file>" text string and click the OK button. It should look similar to Figure 2.10. Now compile the program and look in the program folder. The executable now has the custom icon you selected rather than the standard windows icon.

Caution If you forget to remove the hw.rc after compiling your program, and try to compile a different program in a different folder, you will see an error similar to "Unable to open source file (hw.RC). OBJ file not made." Just remove the resource file reference from the compiler command text box in the settings dialog and the new program will compile correctly.

FreeBasic Help File

You will notice in Figure 2.9 that there is a help file field. You can download a help file for FreeBasic in CHM format from the downloads page of the FreeBasic website. This is a snapshot of the FreeBasic documentation wiki located at the url: http://www.freebasic.net/wiki/wikka.php?FBWiki. The help file may not be as current as the wiki, but it is useful to have for working off line.

Copy the help file to the folder where you installed FreeBasic and point FBIde to this file. Use the button to the left of the Help File text field to browse the location. Once the help file has been set, selecting Help -> from the main menu will load display the help file.

You can also get context-sensitive help for a keyword by placing the cursor on the keyword and pressing F1.

19 Appendix D: Installing FreeBASIC under Linux