Learning Ubercode™

Describes how to install Ubercode and how to write Ubercode programs

Contents

4

1. Getting Started

Welcome to the "Learning to Program" manual. This manual applies to Version 1 of the Ubercode™ computer language released in 2008. This part of the manual explains how to install Ubercode.

1.1 What is in the Package?

Your copy of Ubercode includes the documentation and software in the list below. Also check your computer meets the system requirements before installing.

Package Contents

- Developer Environment with Debugger and Compiler.
- "Learning to Program" Manual (this manual).
- Language Reference Manual (may be on disk).
- Comprehensive on-line help.
- Run Time Library source code.
- Developer Environment source code.
- Over 250 example programs (on disk and in Help files).
- Useful icons and bitmaps.
- Custom Control source code.

System Requirements

- PC with Pentium® processor (or compatible).
- 64 MB memory or higher.
- Microsoft® Windows™ XP, 2000, NT4, ME, 98 or 95.
- VGA or higher resolution screen.
- 50 MB of free disk space.
- CD-ROM, DVD-ROM or access to CD-ROM over a network.

- Mouse or other pointing device.

1.2 How to Install Ubercode

- Put the CD into a CD-ROM or DVD-ROM drive.
- The installer should start automatically. If it does not, click Start – Run, and type **d:setup** in the 'Run' window. The letter **d:** is your CD-ROM drive.
- Confirm whether you accept the license agreement, then choose the directory where Ubercode is installed.
- When the installation is complete there will be a new menu group "Start - Programs - Ubercode". Also the Ubercode icon is shown on the desktop.

The Start - Programs - Ubercode menu has commands to start the Developer Environment, and to open the Help file. There may also be menu options to view the installation notes and other information.

1.3 Contact Details

Ubercode is developed by *www.ubercode.com,* trading as Ubercode Software. We can be contacted at:

- Email - info@ubercode.com
- Website - www.ubercode.com
- Technical support - www.ubercode.com/support

2. Writing your first program

After installing Ubercode you can write a program and compile it to an EXE file. These steps tell you what to do:

2.1 Type in the program

1. Start the Developer Environment by double clicking its icon. The icon is on the desktop.

2. After starting the Developer Environment, the Startup Wizard should appear. Choose "New Text File" and click OK. If the wizard did not appear, wait for the main menu and choose File - New - Text File.

3. Type in the following program. You can use upper or lower case, and you can change the spacing if you want. But don't put spaces in the middle of words. Here's the program:

```
Ubercode 1 class Myfirst

public function main()
var
  MyName:string[*]
code
  MyName <- Inputbox("Myfirst", "What is your name?")
  call Msgbox("Hello "+MyName+"!")
end function

end class
```

The program has two commands, the *Inputbox* and *Msgbox* commands on the lines indented to the right. The *Inputbox* asks for your name, and the *Msgbox* (message box) prints out a greeting message. The program is explained in more detail in the next section "How it Works".

4. Now save the program using the File-Save As command. Save the file as *c:\program files\ Ubercode\ programs\ myfirst*. The Developer Environment prompts you with the correct filename.

5. Now compile the program, using the "Run - Compile this Class" menu command. Wait a few moments for the compiler to finish.

6. If you typed the program as shown it will compile without errors. If there were errors the error message gives the line number of the error, so go to this point and check the code. Check you are typing the double quote character around the strings. You cannot type a single quote twice.

7. After the program successfully compiled run it with the "Run - Start" menu command. The program will show the *Inputbox* window that asks for your name. Type in something and click the OK button. Then the *Msgbox* window will show the name. Click OK to close the *Msgbox* and finish the program.

8. You can now call yourself a programmer as you've successfully installed Ubercode and compiled your first program! This exercise is useful because it also checks everything is installed properly.

2.2 How it Works

You've typed in a program, saved it to disk, compiled it and run it. The program is a list of commands that is understood by the computer. Commands use English words such as *function*, *Inputbox*, *end function*, and symbols such as the plus sign for joining up string text. The compiler converts these commands

to an EXE file. When you run the program, the computer loads the EXE file and runs the commands.

Although the program uses commands in English, it follows a clearly defined structure. This is the *syntax*, and the compiler checks the syntax when it makes the EXE file. If the syntax is wrong, you get a list of errors instead of an EXE file. At the start this may seem a pain, but it's actually a great benefit. If you sell software it means the compiler finds the errors, instead of leaving them in the EXE file for your customers!

The easiest way of learning syntax is to move the cursor onto a command and press the F1 key. The resulting help page includes a description and a simple example.

To understand the syntax of the first program, note it consists of an outer part (the very start and end of the program) and an inner part (everything else). Here is the outer part:

```
Ubercode 1 class ClassName
   ...
end class
```

The command *Ubercode 1 class ClassName* always goes at the start of a program. You choose the *ClassName* as the name of the program, and classes correspond to files, because each class is stored in a single file. Classes always end with *end class* so the compiler knows where to stop.

The rest of the program contains the functions. These carry out the tasks of the program, and this program contains one function:

```
public function main()
var
```

```
  MyName:string[*]
code
  MyName <- Inputbox("Myfirst", "What is your name?")
  call Msgbox("Hello "+MyName+"!")
end function
```

The function is called *public function main()* and the name *main* is special because it makes the class into a main class. A main class can be compiled directly to an EXE file, and when the EXE file starts running, the program starts at *main*. The keyword *public* just before *function main()* ensures *main()* can be called up from outside the class.

The *var* keyword is followed by the memory variables of the program. These store values when the program runs. There is a string variable called *MyName*, which stores text.

The *code* command starts the actual code that runs when the function starts. Code runs in order down the page. Therefore the *Inputbox* command gets the name, then the *Msgbox* command shows the name in a separate window.

Programs can have many functions, and functions can have many commands. The section on "Practical Programming" has examples of more powerful programs.

3. Learning to Program

This chapter is for people who are new to programming. There is a widely held view that programming is very complicated. This is not correct - programming is as simple or as complicated as the programming software you use, and Ubercode has been designed to be as simple as possible.

Most of the complex features in languages such as C++, Java and Visual Basic are unnecessary and have been left out from Ubercode. Modern computers have enough power to automate the tedious detail in earlier languages. Ubercode programmers never have to worry about memory leaks, file handles, invalid pointers, parameter aliasing, header files, pre-processors and macros, DLL compatibilities, framework versions, or Windows API differences between versions.

The rest of this chapter shows the basics of how a computer works and how programs work.

3.1 Computers

The next diagram shows the most important parts of a computer. It applies to nearly all computers made in the last 30 years:
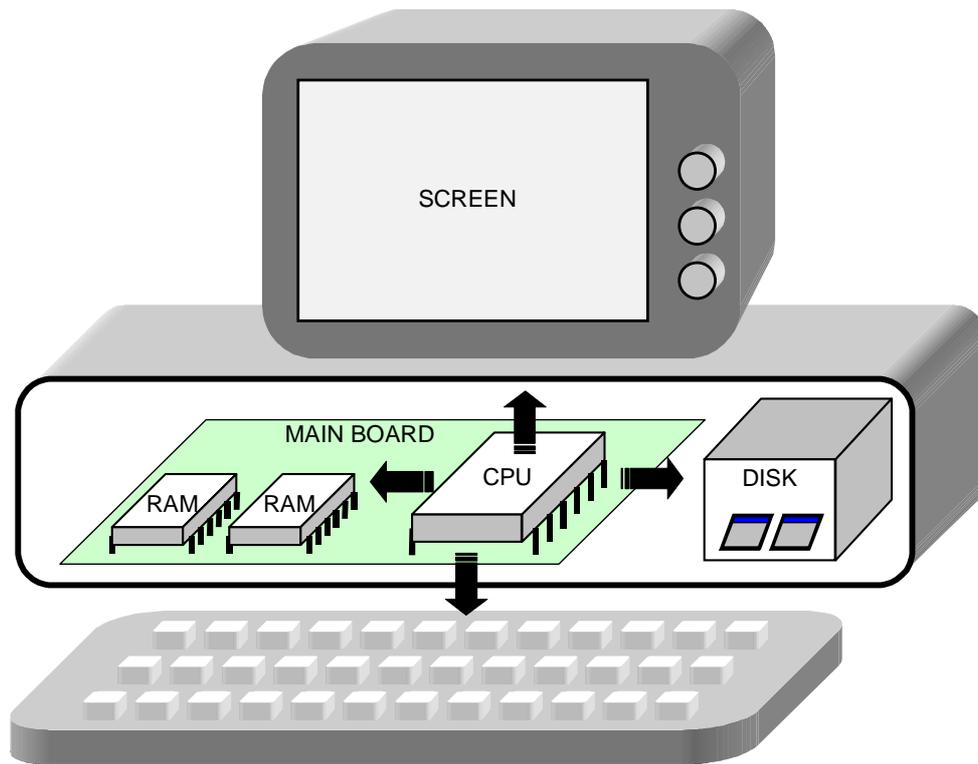
Main board. This is an electronic printed circuit board containing the main components.

RAM. This stands for Random Access Memory, which stores running programs and program variables. The contents of RAM are lost when the computer is switched off.

Disk. This is magnetic storage, used for programs, files, databases and documents. Unlike RAM, the disk keeps its contents when the power is switched off. The small icons in the disk above represent *programs*.

Programs. These contain instructions to be carried out by the CPU. When stored on disk, they are known as *applications*, *EXE files* or *software*. When loaded into memory, they are known as processes, since the CPU is processing them.

CPU. This is the Central Processing Unit. It moves data from one place to another, and does calculations on the data stored in memory. CPUs (also known as microprocessors) are highly complex integrated circuits containing tens of millions of transistors. Typical CPUs are the Intel Pentium® and the AMD Athlon®.

Screen, Keyboard. These interact with programs through the CPU. Other peripherals such as the mouse, sound cards, printers, network adapters etc are common.

Notice that programs are at the heart of the computer. Everything you see on the screen is the result of a program. Microsoft Windows, Microsoft Office, MS-DOS, Unix, games, and accounting software are all just programs that run on the computer. Without programs, a computer is just an expensive pile of electronics. Therefore it's important to understand how programs are made.

3.2 Compilers

From the previous diagram, you saw that programs are stored on disk as EXE (executable) files. When they are run, programs are loaded into memory and processed by the CPU. Inside the computer, the program always exists as binary. Binary is a counting system using base 2, where the only digits allowed are '0' and '1'. This differs from the normal decimal system using base 10, where the digits '0' to '9' are allowed.

You don't need to understand binary to program a computer. But the computer has to use binary because its circuits can only store a '0' or '1'. 0 volts indicate '0', and 3.3 or 5 volts indicate '1'. Everything in the computer's memory and disk storage uses binary (the disk uses magnetism instead of voltage).

Programming in binary is very complex, for example the following is part of a program that copies a short message to the screen:

```
1011 1010 0010 0000 0000 0000
1011 0100 0000 1001
1100 1101 0010 0001
```

Programmers prefer English text to binary; therefore they need a system to convert written text into binary. This is done using a program called a compiler. The compiler processes the written text commands and produces equivalent binary code. For example the compiler converts the command *PRINT "Hello world"* into binary code, then stores the binary in an EXE file. The binary code tells the CPU to copy "Hello World" to the screen. The next diagram shows this:
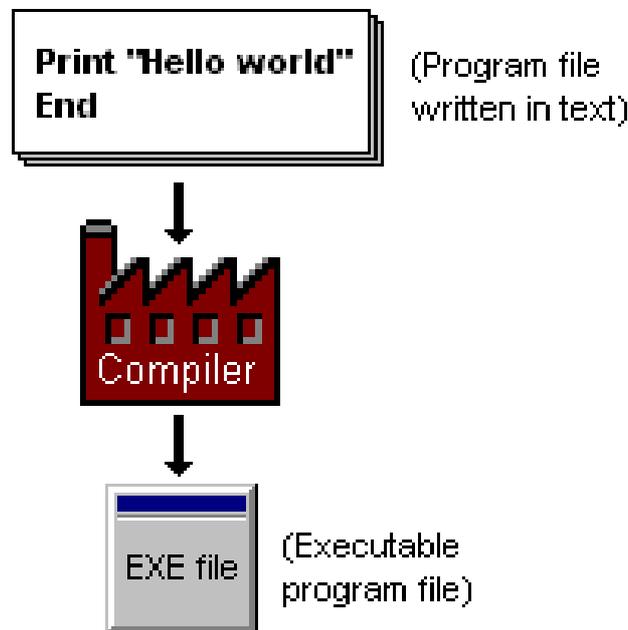
Understanding this diagram is an important step to understanding programming. Programmers use written text because binary is too complex. The computer only understands

binary because that's how the electronics work. The compiler converts the text commands into the binary equivalent. All computer languages and software development packages follow this process.

Although compilers are sophisticated, they are not all-powerful. For example *"Write me an accounting program"* is too vague. Instead, compilers understand a strictly limited set of commands. The set of commands is a *computer language*, and each language has its own syntax. In effect the compiler promises "if you give me a valid program, I'll convert it into binary for you".

Compilers are internally designed to process a single language, and to generate binary code for one type of computer. When talking about compilers, you need to specify the language and the type of computer. For example Microsoft C++ converts C++ code into binary for Windows. Languages with compilers include C, C++ and Java (powerful but very complex), and different versions of Basic.

Basic was invented in 1964 as a simple and powerful language. After Windows became popular Visual Basic and VB.NET replaced Basic, but these modern versions are nearly as complex as C++.

IBM invented the first compiler in the 1950s, and the first computer language was Fortran. The invention of compilers and languages is the reason for the improvements in computers and software since the 1950s.

3.3 Programs

We've just seen how compilers are essential when writing programs. They allow programmers to use words and symbols instead of '1's and '0's. The following diagram shows the (idealized) approach used by programmers:
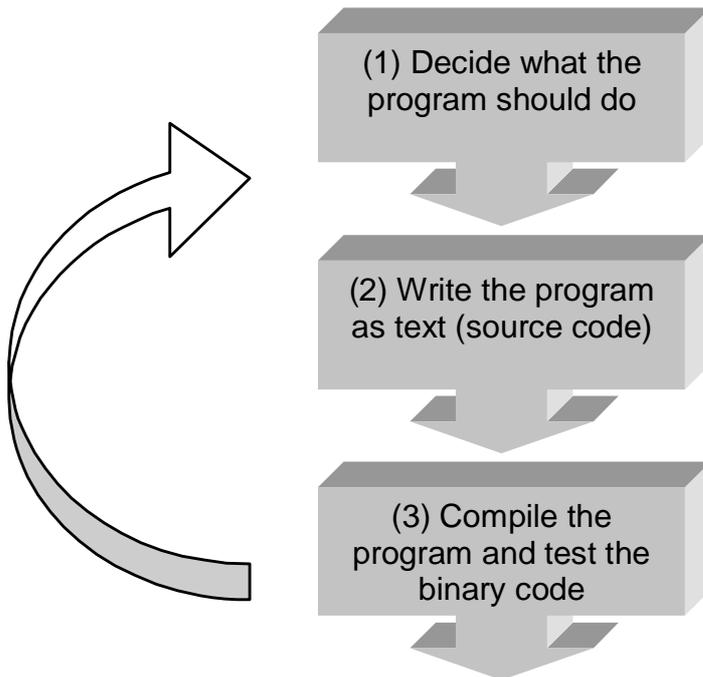
In step (1), decide what the program does. This is not an elaborate description, for a small program it should be just a few sentences. Time spent on the description is repaid when testing and debugging. This information can go at the top of the program, in the form of a comment, so it won't get lost. The only time descriptions need to be separate is when they are reviewed as a separate document.

In step (2), write the actual program using the language processed by the compiler. In this form, the program is known as *source code*.

In step (3), compile the program and test the binary code. If the program had syntax errors or used unrecognized commands, the compiler cannot create the binary code and will show an error message instead. Errors must be fixed, so go back to step (2) and modify the source code.

This process is the " Edit - Compile - Run" cycle and is common to all types of programming. Modern languages hide the details behind elaborate interfaces and fancy terminology, but the basics are the same.

4. Practical Programming

The programs in this chapter show useful programming tasks, such as finding averages of numbers, reading in text, and printing. All the examples should compile and run without error - also they assume you've installed Ubercode. If not, follow the instructions in Chapter One.

4.1 'Averages' program

The 'Averages' program inputs one or more numbers, and then shows the average. The program shows an error message if the inputs are invalid.

Write the source code

The easiest way of writing the program is by typing directly into the Developer Environment. Start the Developer Environment, and then press Ctrl+N for a new text file. Or you can load the *c:\program files\ Ubercode\ examples\ math1\ averages.cls* file.

The first step is to write a function that finds the average of one or more numbers. Type in the following function:

```
function Average(in a:Tnumbers[*:*] out av:real)
var
  i:integer(0:MAXINT)
  total:real(0:MAXINT)
code
  av <- -1.0
  if Ubound(a) > 0 then
    for i from Lbound(a) to Ubound(a)
      total <- total + a[i]
    end for
    av <- total / (Ubound(a)-Lbound(a)+1)
  end if
end function
```

Next, we need a routine to read in one or more integers and return them as an array. The integers are returned in an array sized from 1 to the number of integers. If the integers are incorrect or nothing is entered, the returned array has the dimensions (-1,-1):

```
function GetNumbers(out a:Tnumbers[*:*])
var
  InputStr:string[*]
  OK:boolean
code
  InputStr <- Inputbox("Averages", "Enter Numbers")
  call Val(InputStr, a, OK)
  if not OK then
    call Redim(-1, -1, a)
  end if
end function
```

Next, there is a *public function main* that calls the other functions. The function is called *main* to make sure it runs first, and it is *public* to make it available outside the class. The main function detects an error by checking for *av = -1.0*. This will happen if nothing was entered, or if text was entered instead of integers. If the average was calculated correctly, it is displayed:

```
public function main()
var
  av:real(0:MAXINT)
code
  av <- Average(GetNumbers())
  if av = -1.0 then
    call Msgbox("No inputs")
  else
    call Msgbox("Average="+Str(Fix(av)))
  end if
end function
```

These three functions (*Average*, *GetNumbers*, *Main*) form the main part of the program. We have to write a main class

(equivalent to a program) using these functions. Also we declare a type that stores an array of integers. Here is the main class (the italics represent the functions you just typed):

```
Ubercode 1 class averages

type Tnumbers[*:*]:array[*:*] of integer(0:MAXINT)

function Average...

function GetNumbers...

public function main...

end class
```

That completes the program. If you've just typed it in, save it and print it if possible. The entire program should read:

```
// The "Averages" program inputs one or more
// numbers, then shows the average. The
// program shows an error message if the
// inputs are invalid.
Ubercode 1 class averages

type Tnumbers[*:*]:array[*:*] of integer(0:MAXINT)

function Average(in a:Tnumbers[*:*] out av:real)
var
  i:integer(0:MAXINT)
  total:real(0:MAXINT)
code
  av <- -1.0
  if Ubound(a) > 0 then
    for i from Lbound(a) to Ubound(a)
      total <- total + a[i]
    end for
    av <- total / (Ubound(a)-Lbound(a)+1)
  end if
end function

function GetNumbers(out a:Tnumbers[*:*])
```

```
var
  InputStr:string[*]
  OK:boolean
code
  InputStr <- Inputbox("Averages", "Enter Numbers")
  call Val(InputStr, a, OK)
  if not OK then
    call Redim(-1, -1, a)
  end if
end function

public function main()
var
  av:real(0:MAXINT)
code
  av <- Average(GetNumbers())
  if av = -1.0 then
    call Msgbox("No inputs")
  else
    call Msgbox("Average="+Str(Fix(av)))
  end if
end function

end class
```

At this point the program exists as a source code file on disk, and you could come back tomorrow and it would still be there. But the program can't run yet, since it doesn't exist as binary. The next step creates the binary code.

Compile the program

The compiler converts the program into binary code. Press the compile button ▓ and wait for the compiler to finish. If there are any errors check your program against the printout and re-compile it. After it compiled successfully, press the Run ▶ button. This loads the binary code into memory and makes the CPU process the binary code.

The windows and calculations that occur now are done by the binary code of your program. Enter some numbers (for example 30 40 50) and check the result:
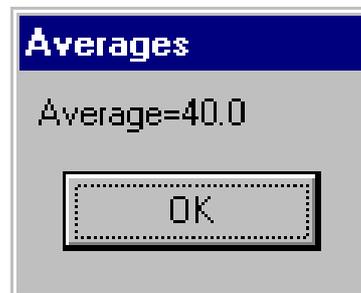
Click OK to close the program.

This example shows the "Edit - Compile - Run" cycle in action. You described the program with a comment at the top; you typed it in and compiled it. You fixed the errors, then ran it and tested it. The result is an EXE file that could be copied and run on any Windows computer anywhere in the world.

4.2 'Read Text' program

This program shows how to read in a text file. It prompts you for a file name, then it displays the file in a *Listbox*. Type in the source code as shown, or load the *c:\program files\ Ubercode\ examples\ files1\ files1.cls* file:

```
Ubercode 1 class files1

public function main()
var
  filename:string[*]
  textstring:string[*]
code
  filename <- Openfiledialog("Read Text", 0, "*.txt")
  if filename /= "" then
    call Loadfile(filename, FILE_TEXT, textstring)
    call Listbox("Read Text File", "", textstring)
```

```
   end if
end function
end class
```

When the program starts running, Windows calls *public function main()* as the first function. *Main* declares two local variables, *filename* and *textstring*. The filename stores the name of the text file and *textstring* stores the text loaded from the file.

The *Openfiledialog* command in *main* prompts you for the file name. This is returned in the *filename* variable, unless you press Cancel in the dialog box in which case *filename* returns an empty string. The program then tests *filename* to make sure you did enter a name, and then it calls *Loadfile* to load the text file into the string variable *textstring*. *Textstring* stores the actual text of the file.

Finally the *Listbox* command displays the text string in a scrolling list. After viewing the text, click OK to end the program.

To run the example, first type it in and save it, or load the example. Then compile it, wait for the compiler to finish, and run it.

4.3 'Print1' Program

This program shows how to copy text to the printer. Type in the class shown below, or load the *c:\program files\ Ubercode\ examples\ print1\ print1.cls* file:

```
Ubercode 1 class print1

private function Printme(in text:string[*])
var
   count:integer(0:MAXINT)
```

```
code
  call Startprint(Sysprinter)
  call SetFontname(Sysprinter, "Times New Roman")
  call SetFontsize(Sysprinter, 14)
  call SetFontitalic(Sysprinter, True)
  for count from 1 to Strcount(text)
    call Drawtext(Sysprinter, Strline(text,count))
  end for
  call Endprint(Sysprinter)
end function

public function main()
var
  poem:string[*]
code
  poem <- "Roses are red" + NL +
          "Violets are blue" + NL +
          "Some poems rhyme" + NL +
          "But this one doesn't." + NL
  call Printme(poem)
end function
end class
```

The program works as follows. The code that does the printing is put in a separate function *Printme* to make the program easier to understand. In general function *main* should contain as little code as possible, to make it easier to change the program as it develops. The most important lines in function *Printme* are:

```
  call Startprint(Sysprinter)
  call Drawtext(Sysprinter, text)
  call Endprint(Sysprinter)
```

Startprint initializes the printer and displays a window that shows the printer's progress. *Drawtext* copies some text to the printer, moving down to the next line if *text* ends with a new line character (the NL character). You could call any drawing routine here, such as *Drawpicture* etc, but this program keeps to printing text for now. Finally *Endprint* closes the printer and removes the printer progress window.

As you can see from the code, function *Printme* works by calling *Startprint*, then sets a more interesting font which is Times New Roman 14 point italic, then sets up a loop to get each line of the string and calls *Drawtext*. When *Printme* has looped through all the lines in the text string, it calls *Endprint*. You could easily include *Printme* in other programs as a routine for printing text.

The printer used by *Printme* is *Sysprinter*, which is the system default printer. This is set up under Windows by Start - Settings - Printers, then right-clicking on the printer you want to use as the default.

Moving on to function *main*, this makes up a string containing a short poem, then it calls *Printme* to print the text.

After typing in the class or loading it from the file, you can run it. Compile the program first, wait for the compiler to finish, then run it.

5. Testing and Debugging

This section describes how to fix errors. Errors can happen when compiling a program, when running it or if it produces wrong output.

5.1 Types of Error

Compile time errors

Compile time errors are caused by a class that has incorrect syntax or that uses language commands incorrectly. The compiler detects the error, and after it finishes, the Developer Environment shows a window with all the errors. You must fix all compile time errors before running a program.

To fix the error change the class source code near the line of the error. The error message includes the class name and the line of source code that was wrong; also it includes the error number and a hint to explain the error.

Click on the error description and press the F1 key to read more details on the error. Also you can double click the error description to jump to the line of the error.

Run time errors

These occur if a running program has a problem that prevents it continuing, such as not enough memory, missing files, or reading and writing outside the limits of an array. When a run time error takes place the program displays a message and stops. The message includes the error number, class name and line number of where the error happened.

To fix run time errors use the Developer Environment to load the class that caused the error, and go to the line of code that went wrong. You can get help on the error by searching the help system for the error number.

If it's not clear what caused the error run the program under the debugger. Use the Tools - Options - Compiler command to bring up the compiler options dialog. Click on "Full Debugging", then click OK to close the dialog. Now recompile the application using the Run - Full Rebuild EXE File command. After it has compiled use the Run - Start command to run the program.

When the program starts use the same commands that caused the error previously. This time instead of seeing an error message, the debugger pops up at the point of the error. Use the Watch button to look at the values of variables, or the Call stack button to see the sequence of calls by which the program reached the error. By checking the values are correct and seeing if the program flow took the correct path, you can see where the error is. Use the debugger to halt the program, edit the source code and recompile the program.

Logic errors

These are errors in a running program that do not cause a run time error, but instead the program produces incorrect output or gets stuck in a loop.

To fix these, use the Run - Start in Debugger command. When the debugger starts, put breakpoints near where the program started to go wrong, and in other places you're not sure about. After setting the breakpoints, click on "Run" and wait for a breakpoint. When it reaches a breakpoint, step through code

and use the "View Watches" and "View Calls" buttons to check the program is running as you expect. Again you need to make sure program values are correct and the program is taking the correct path.

After checking the code you may see the error, or have some idea of extra code to improve the program logic. Halt the program, edit the source code and recompile again. This is the "Edit - Compile - Run" cycle.

5.2 Bugslayer in Action

After all this theory it's time to catch some bugs. The following class is intended to print out the squares of the numbers 1 to 10, but it has some bugs. Type in the class as shown below, or load it from *c:\program files\ Ubercode\ examples\ bugcity\ bugcity.cls.* The class below includes the bugs, but if you can see them don't fix them yet!

```
1.   Ubercode 1 class Bugcity
2.
3.   type
4.     NumArray[*:*]:array[*:*] of integer(0:MAXINT)
5.
6.   public function main
7.   var squares:NumArray[*:*]
8.   code
9.     result <- Str(squares)
10.    call CalcSquares(squares)
11.    call Msgbox(result)
12. end function
13.
14. function CalcSquares(in squares:NumArray[*:*])
15. var count:integer(0:MAXINT)
16. code
17.    for count from 1 to 10
18.      squares[count] <- Sqr(count)
19.    end for
20. end function
```

```
21.
22.   end class
```

Save the class, then compile it. There should be a nasty outbreak of compile time bugs as follows:

1. bugcity.cls(7): Error 2002 : Unexpected symbol. *Var is an unexpected symbol here because main is a function name and should be followed by round brackets.*

2. bugcity.cls(9): Error 2561 : Identifier is not declared. *Result should be declared as a string variable here. All identifiers must be declared in Ubercode.*

3. bugcity.cls(10): Error 2561 : Identifier is not declared. *The CalcSquares function is not declared. This error is more subtle because the function is in the class, but it is after function main. To fix this error we have to put a public prototype (function header) of CalcSquares before function main. Also if CalcSquares is declared using a function prototype, any types it uses (NumArray) must also be declared public.*

4. bugcity.cls(11): Error 2561 : Identifier is not declared. *Again this is the undeclared result variable we know about.*

5. bugcity.cls(18): Error 2530 : Identifier is not allowed here. *The squares array cannot be used as a left hand value because it is an **in** parameter which is not modifiable. To fix this squares has to be changed to an **inout** parameter.*

6. bugcity.cls: Error 2607 : Function main is not declared properly. *This is another message that happens because main was not followed by round brackets.*

To get help look up the error number in the on-line help system. The comments in italics above describe how to fix the errors, so make changes as described. The corrected class should look like this with the changes shown in bold (note the round brackets after *main*):

```
Ubercode 1 class Bugcity2

public type
  NumArray[*:*]:array[*:*] of integer(0:MAXINT)

public function CalcSquares
                (inout squares:NumArray[*:*])

public function main()
var squares:NumArray[*:*]
    result:string[*]
code
  result <- Str(squares)
  call CalcSquares(squares)
  call Msgbox(result)
end function

public function CalcSquares
                (inout squares:NumArray[*:*])
var count:integer(0:MAXINT)
code
  for count from 1 to 10
    squares[count] <- Sqr(count)
  end for
end function
end class
```

The type *NumArray* and function *CalcSquares* have been moved to before function *main*, so they are in scope. The first occurrence of *CalcSquares* is called a prototype, because it consists of just the function header. Prototypes, and types used by prototypes, must be public, therefore *NumArray* and *CalcSquares* use *public* in their declaration.

Also we have added the empty parameter list following *main*, and we have declared the *result* string. After making these changes the compile time errors are fixed and we can compile the class successfully.

But after compiling Bugcity and running it we get the following run time error (the line number may vary slightly):

```
Error 602

Array reference out of bounds
File BUGCITY2, function calcsquares, line 23

            OK
```
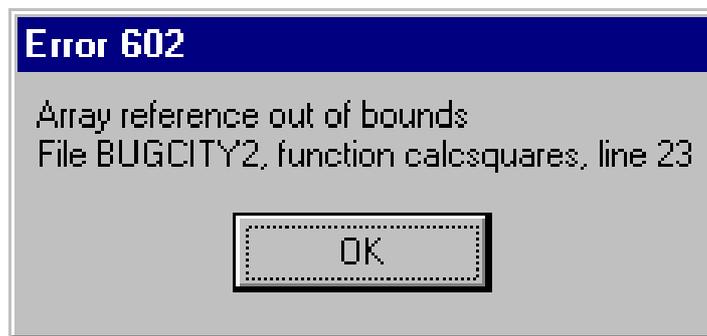
This error happens on the following line in CalcSquares:

```
squares[count] <- Sqr(count)
```

because the array reference *count* is outside the array bounds. Note the array is indexed from 1 to 10, but the array *squares* passed to *CalcSquares* from *main* was declared in *main* as a resizable array (using [*:*] notation):

```
var squares:NumArray[*:*]
```

Because *squares* does not have bounds 1 to 10, this causes the array reference error. To fix the bug change the declaration of *squares* so it has bounds of 1 to 10, as follows:

```
var squares:NumArray[1:10]
```

Then recompile the program and re-run it. Now the program will run without a run time error but instead of printing the correct values, it shows the following:

The number zero is printed ten times, instead of the squares of 1 to 10. This is a logic error, because the program compiles and runs without crashing, but the results are not as expected.

The debugger is useful for fixing logic errors. Use the Tools - Options - Compiler command, click next to "Full debugging" and click OK. Then use "Run - Start in Debugger" to recompile the program and start the debugger at function *main*. Now we want to find why the *squares* array is full of zeros. By looking at the code we can see that when *CalcSquares* returns, the array should have the correct values. If we run the program to this point we can check the array. Put a breakpoint on the line:

```
call Msgbox(result)
```

which is the line of code after *CalcSquares* returns. To set the breakpoint, double-click the line of code in the debugger. Then click the "Run" button to run the program. When the debugger comes active again, click the "View Watches" button to see whether *CalcSquares* did its job. You should see the following in the Watches window:

```
squares : Array = {1,4,9,16,25,36,49,64,81,100}
  result : String = "{0,0,0,0,0,0,0,0,0,0}"
```

Very interesting! It shows the *squares* array contained the correct values all the time, but it was the *result* string that is wrong. Look at the code in *main* and notice that we are converting *squares* to the *result* string <u>before</u> calculating the array values. To fix this move *Str(squares)* to after the call to *CalcSquares*. After fixing the error *main* should look like this:

```
public function main()
var squares:NumArray[1:10]
    result:string[*]
code
  call CalcSquares(squares)
  result <- Str(squares)
  call Msgbox(result)
end function
```

Halt the debugger, make the changes and recompile and re-run the program, and click Run when the debugger starts up. Now you should finally see the program working properly, and it should show the following:

To make sure the fix worked, put back the breakpoint after the call to *CalcSquares* and check both *squares* and *result* have the correct values. After fixing logical errors, it's a good idea to re-run the debugger and make sure the fix has worked.

5.3 Getting Extra Help

When dealing with errors, use the on-line help. Activate help
from the Developer Environment using the Help - Contents
command, then in the Help window click the "Index" button
and type in the error number. This works for compile time and
run time errors.

The on-line help is also useful if you're having problems with a
particular command. Look up the command in the help system,
as all the language commands have an example program that
shows the correct usage.

If none of the above help, use the Technical Support website
www.ubercode.com/support, or refer to the contact details
in the Introduction to this manual.

6. Reference

This section includes the Technical Specifications of Ubercode, a glossary of terms and an ASCII table.

6.1 Technical Specifications

Developer Environment

- Produces standard Windows applications (EXE files)
- EXE files run under all supported versions of Windows
- Source code is stored in classes
- Window layout editor and Integrated Debugger
- Compiles single and multi class applications
- Supports Windows XP, 2000, NT4, ME, 98, 95
- Automatic DLL versioning

Control Flow

- Structured control flow (if, else, end if)
- Multi-way control flow (select, case, end select)
- Loops (for, while, loop, exit when, end loop)
- Data file access (for each, iterate)
- Classes, functions and interfaces
- Event driven code with callbacks

File Types

- DBF (dbase), XML and CSV for database applications
- ICO and BMP files for images
- TXT files
- INI files for configuration data
- WAV files for multimedia
- HLP files for Windows help

- RC and DLG files for window layouts
- Free-format binary files for general applications

Data Types

- Logical values (true, false)
- Integers, fixed point and floating point numbers
- Fixed and variable length text strings
- Records (user defined types)
- Sets of integers and Safe Arrays
- Lists and tables (in memory and for file access)
- Abstract Data Types (for Object Oriented Programming)

Visual Object Types

- Edit object (single and multi line)
- Radio buttons, Check boxes (for selecting options)
- Push buttons and Bitmap buttons
- Scroll bars and Progress bars (changing and displaying values)
- Combo box, List box and List box with icons
- Pictures and Icons (BMP or ICO format)
- Group box, Label text
- Shapes (coloured borders and interior)
- Menus, Clipboard, Printer, Screen

Window Types

- Message box, Input box, List box
- Dialog box object
- Scrolling Edit window object
- Common dialogs (Open, Save As, Print, Font, Color)

6.2 Glossary

Array. An array is a variable that stores one or more items of the same type.

Boolean. A data type that is able to store True or False values.

Compiler. This converts Ubercode classes into EXE files and is called automatically from the Developer Environment. You can also call it up directly, to compile classes without using the Developer Environment.

Debugger. This is an add-on component of the Developer Environment. It monitors programs when they run, and allows you to step through them line by line using a graphical interface.

Developer Environment. An application used by software developers, which allows you to type in, edit, print, run and test programs, then convert them into EXE files.

Dialog box. A dialog box (also known as a *Form*, or a *Window*) is a rectangular area on the screen with control objects for entering or displaying data. Controls include push buttons, scroll bars and more.

Dialog editor. This is a graphical editor for designing dialog boxes. It lets you add controls, move them to different places, and set their properties. The Ubercode Developer Environment includes a dialog editor. Dialog editors are also called Form editors, Resource editors, or Layout managers.

Inputbox. A modal dialog used for inputting a string.

Integer. A data type which stores numbers without fractional parts.

Iterator. An iterator is a loop which processes all the items in a list or table. Refer to the *For each* or *Iterate* command in the help file.

Listbox. A listbox is a scrolling list of items on the screen. Listboxes may also contain small pictures to the left of the text. Refer to the *Iconlist* object in the help file.

Loadfile. A command which reads in an entire file using a single instruction.

Modal window. When a modal window is active, it locks out all other windows in your application, although you can still switch to other applications. The *Show* method is used for displaying modal and modeless windows, and *IsModal* tests whether a window is modal.

Modeless window. When a modeless window is active, you can switch to other windows in your application by clicking on them. When different windows in your application are activated, they trigger events that run code in your program.

Msgbox. A modal dialog used for displaying a string and some push buttons. The string can have multiple lines, also you get to choose the push button labels.

Openfiledialog. A modal dialog provided by Microsoft for opening files. The Openfiledialog allows you to change to different directories, and to choose one or more files.

Parameters. When inside a function, this is the name given to the function's in, inout and out parameters.

Postcondition. This applies to a function and is an expression using the function's parameters which is tested just before the function returns. It checks the function produces correct outputs.

Precondition. This applies to a function and is an expression using the function's parameters which is tested just before calling the function. It checks the function is passed correct inputs whenever called.

Resource editor. See dialog editor.

String. This is a very useful data type that stores text. Strings can also store multiple lines of text using the new-line (NL) character as an end-of-line marker.

6.3 ASCII code table

The next table shows the ASCII character set. Dec and Hex are the ASCII value and Chr is the corresponding character. If no character is shown, there is no printable ASCII value. The character denoted SP indicates a space, and Cur denotes the national currency symbol.

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
0	0		32	20	SP	64	40	@	96	60	`
1	1		33	21	!	65	41	A	97	61	a
2	2		34	22	"	66	42	B	98	62	b
3	3		35	23	#	67	43	C	99	63	c
4	4		36	24	Cu	68	44	D	10	64	d
5	5		37	25	%	69	45	E	10	65	e
6	6		38	26	&	70	46	F	10	66	f
7	7		39	27	'	71	47	G	10	67	g
8	8		40	28	(	72	48	H	10	68	h
9	9		41	29	)	73	49	I	10	69	i
10	A		42	2A	*	74	4A	J	10	6A	j
11	B		43	2B	+	75	4B	K	10	6B	k
12	C		44	2C	,	76	4C	L	10	6C	l
13	D		45	2D	-	77	4D	M	10	6D	m
14	E		46	2E	.	78	4E	N	11	6E	n
15	F		47	2F	/	79	4F	O	11	6F	o
16	10		48	30	0	80	50	P	11	70	p
17	11		49	31	1	81	51	Q	11	71	q
18	12		50	32	2	82	52	R	11	72	r
19	13		51	33	3	83	53	S	11	73	s
20	14		52	34	4	84	54	T	11	74	t
21	15		53	35	5	85	55	U	11	75	u
22	16		54	36	6	86	56	V	11	76	v
23	17		55	37	7	87	57	W	11	77	w
24	18		56	38	8	88	58	X	12	78	x
25	19		57	39	9	89	59	Y	12	79	y
26	1A		58	3A	:	90	5A	Z	12	7A	z
27	1B		59	3B	;	91	5B	[	12	7B	{
28	1C		60	3C	<	92	5C	\	12	7C	|
29	1D		61	3D	=	93	5D	]	12	7D	}
30	1E		62	3E	>	94	5E	^	12	7E	~
31	1F		63	3F	?	95	5F	_	12	7F	