Aurora Tutorial

By
Paul Turley

Part 1. Getting started with Aurora

The Start

The hardest part when learning a new programming language is knowing where to start. This tutorial is aimed at the new Aurora user who has used other programming languages in the past. In Part 1 we will go over basic program structure, variable types, console I/O functions, loops and conditional statements.

Our first step is to launch Aurora and create a new source file. Do so now by clicking on the **File** menu and selecting **New -> Aurora Source File**

Save this file to disk by clicking on the **File** menu and selecting **Save As..**. Choose and appropriate file name such as "Tutorial1.src".

Every Aurora program has a starting point, where your code begins execution, and that is a subroutine named **main**. In this tutorial we will use the terms "subroutine" and "function" interchangeably since there is no difference between the two in Aurora.

The main subroutine must be visible to the outside world, and in Aurora this is done by using the GLOBAL keyword. Type the following in the blank source file:

```
global sub main( )
   {
   }
```

That is the beginning of any Aurora program.

Remarks can be inserted in the program by using:

```
// This is a of a remark.  It exists on a single line only.

/* Multiple line remarks may be indicated
    By starting with the slash & asterisk
    and terminating the remark or notes with an asterisk & slash */
```

It is optional whether braces are indented or not. Indenting the braces make for easier reading, but for brevity in a document as this, it is sometimes a necessity not to indent them.

CONSOLE Basics

The Windows console, or shell, provides text based input and output facilities. Aurora supports text based programming with the built in console library. Add the following lines, in blue, to your source window.

```
global sub main( )
  {
  print("Hello from Aurora");
  print("press any key to close");
  While GetKey() = "";
  }
```

Note that each code line is terminated with a semi-colon.
Now let's compile the program and test it out.
Click on the **Build** menu and select **Build Single**. This tells Aurora that you want to compile the visible source file, and it is the only source file for your program.
When the "Executable Options" dialog appears use the "Target" pulldown box to choose "Console EXE". Check the box next to "Execute after creating" and finally click on the "Create" button.

If we did everything correctly you should see the console window appear with the text from the program printed in it.
Press any keyboard key to end the program.

The **print** function can output any data type to the console. Print will automatically send a newline to the console unless you tell it not to by adding a comma to the list of items to output. Add the following lines, in blue, to your source window.

```
global sub main( )
  {
  print("Hello from Aurora");
  print("What is your name? ",);
  name = readln();
  print("Hello ", name, ", Nice to meet you");
  print("press any key to close");
  While GetKey() = "";
  }
```

The **readln** function reads a line of text from the console and waits for a the return key to be pressed. You can also read a single character from the keyboard using the **GetKey** function. GetKey returns the pressed key in either Ascii or Raw format. Raw format allows reading special keys such as the function keys. Let's change out program to wait for the F6 key to be pressed before we close. Change the following lines, in red, in your source code window:

```
global sub main( )
  {
  print("hello from Aurora");
  print("What is your name? ",);
  name = readln();
  print("Hello ", name, ", Nice to meet you");
  print("press the F6 key to close");
  While GetKey(true) != "\x75";
  }
```

Compile and run the program. "\x75" creates a one byte string with the character 0x75 which is the virtual key code for F6. A list of virtual key codes can be found at the end of this tutorial.

Now that we have the basics of console input and output we can move on to our next subject. Variables.

Variables and intrinsic data types

A variable is a temporary storage location for information in your program. Think of it as memory for the data used in your program. Variable names can be any identifier you choose as long as they do not conflict with any reserved word or constant.

Aurora supports local, source global and program global variables.

A *local variable* is a variable that is defined within a subroutine and is only accessible from within that subroutine.

A *file global variable* is a variable that is defined outside of a subroutine and is accessible by any subroutine located in that file.

A *program global variable* is a variable that is defined outside of a subroutine and marked with the GLOBAL keyword. A program global variable can be accessed by any source module in your program.

An intrinsic data type is a fancy name for what built in variable types a language supports.
Aurora supports the following intrinsic types:

BYTE - Can store a single character or an integer number from 0 - 255 (if unsigned)
WORD - 2 bytes, An integer number from 0 - 65535 (if unsigned)
INT - 4 bytes, An integer number from 0 - 4294967295 (if unsigned)
INT64 - 8 bytes, An integer number from 0 - 18446744073709551615 (if unsigned)
FLOAT - A single precision floating point number.
DOUBLE - A Double precision floating point number.
POINTER - An address
STRING - A string of characters up to a length of 255
DSTRING - A dimensionable string that can contain any number of characters.

The keyword **UNSIGNED** can be used with the integer types BYTE, WORD, INT and INT64 to define an unsigned variable. The default is a signed variable.

A variable can be defined by assignment, using the DEF statement, or using the type-name syntax. Aurora supports the auto-definition of variables by default unless turned off by using

```
#autodefine "off"
```

Auto-definition creates and specifies a variable by assigning it a value. In our example program the variable 'name' was created and defined by assigning the return from readln(). In this case it was a STRING variable.

Here are some examples of defining variables:

```
a = 1;
name = "John";
float flValue;
def dbValue as DOUBLE;
```

The auto-definition of variables is a convenience but should not be relied upon if you are not sure what kind of variable will be created, or if you prefer to define all of your variables ahead of time. Auto-definition of variables can also create subtle bugs that are hard to track down if you have local and global variables with the same name.

Let's change our program to use a few variables. Add the lines, in blue, in the source code window.

```
global sub main( )
  {
  int age; //An integer variable
  float money; //a floating point variable
  print("hello from Aurora");
  print("What is your name? ",);
  name = readln();
  print("Hello ", name, ", Nice to meet you");
  print("How old are you? ",);
  age = StrToNum(readln());
  print("How much do you make a year? $",);
  money = StrToNum(readln());
  print("So you are ",age," years old and make $",money);
  print("You need a raise");
  print();
  print("press the F6 key to close");
  While GetKey(true) != "\x75";
  }
```

Compile and run the program to see the results.

The variables defined in the program are local variables. They are only accessible within the 'main' subroutine. Which is the only subroutine in our program. It is important to remember that local variables will contain random information until they are assigned a value. Do not depend on the contents of any variable before they are assigned. Aurora supports a single step definition and assignment using the type-name syntax:

```
double dbValue = 1.04567;
string strName = "A tale of two computers";
```

Global variables cannot use the single step definition and assignment. A global variable exists outside of the path of executing code so the assignment will be ignored.

Arrays

Arrays of up to three dimensions are supported by Aurora. An array of more than three dimensions can be created using memory allocated by the NEW function. For this tutorial we will concentrate on normal arrays.

Defining an array is done with brackets:

```
int myArray[10,100];
double prices[500];
```

Arrays are accessed by using brackets and an index to the array location.
Indices in Aurora are zero based, which means if you want to access the 10th element of a dimension the index is 9:

```
current_price = prices[499];
```

Arrays can be initialized with a list of values, separated by commas.

```
prices = 1.5,300.50,40.25,77.99,175.65,199.33,1000.99,200.32,123.55,763.33 ;
```

Let's change our program and add an array.
We will also use a loop, which hasn't been covered yet, but keep following along until we get there.
Add the lines, in blue, in the source code window.

```
global sub main( )
  {
  int age; //An integer variable
  float money; //a floating point variable
  string colors[8]; //an array
  colors = "red", "blue", "green", "yellow",
  "purple", "burgandy", "orange", "violet";
  print("hello from Aurora");
  print("What is your name? ",);
  name = readln();
  print("Hello ", name, ", Nice to meet you");
  print("How old are you? ",);
  age = StrToNum(readln());
  print("How much do you make a year? $",);
  money = StrToNum(readln());
  print("So you are ",age," years old and make $",money);
  print("You need a raise");
  print();
  print("My favorite colors are: ",);
  for(x = 0; x< 8; x++)
    {
    print(colors[ x ]," ",);
    }
  print();print();
  print("press the F6 key to close");
  While GetKey(true) != "\x75";
  }
```

NOTE:
Multiple statements may be used on a single line providing each statement is separated by a semi-colon.

In addition to intrinsic types Aurora allows creating new variable types through the use of the #typedef pre-processor command.
Using #typedef you can create aliases to types and new type names. It is common in programming to create aliased type names to represent a specific type of data and to improve readability. There is no difference between using the original type and its alias.

Examples:

```
#typedef UINT unsigned int
#typedef HWND UINT
#typedef HANDLE UINT
#typedef BOOL int
#typedef CHAR byte

HANDLE hFile = 0;
BOOL bDone = false;
```

NOTE:
Pre-processor commands such as #typedef, #include, and #autodefine etc. do not need a terminating semicolon as they do not represent code, but a command to the compiler itself.

User defined variable types

Aurora supports the creation of user defined variable types, or *structures* as they are commonly called. A structure is a collection of variables tied together by name. Each variable within a structure is called a member. The **struct** statement is used to begin the definition of a structure and the member variables are surrounded by braces.

An example structure:

```
struct FLOATRECT
  {
  float left;
  float top;
  float width;
  float height;
  }
```

The structure name becomes a new type of variable that you can define in your program:

```
FLOATRECT flRect;
```

When a structure variable is created, all of the member variables are created and accessed by using either the dot '.' or dereference '->' operators.
The dereference operator is used when you have a pointer to the structure., such as when it is created by NEW.
As an example, using the FLOATRECT structure above, would look like:

```
flRect.left = 100.0;
flRect.top = 50.5;
flRect.width = 25.5;
flRect.height = 25.5;
```

While we have not yet covered memory allocation with Aurora, it is important to know the syntax of using pointers, so a short example is warranted:

```
FLOATRECT *pRect = NEW(FLOATRECT,1);
pRect->left = 100.0;
pRect->top = 50.5;
pRect->width = 25.5;
pRect->height = 25.5;
```

Pointers and the dereference operator will be covered in more detail in Part 2 of this tutorial.

Constants, literals, and type modifiers

Constants are identifier to numeric mappings that cannot change value. They are similar to variables in that they are referred to by name but don't actually use memory or generate any code.
The compiler substitutes a constant identifier with its numeric identity at compile time.
Aurora supports two constant definition statements, CONST and #define. Both statements can create numeric constants, #define can also create pre-processor definitions.

Examples:

```
#define WM_USER 0x400
#define APP_CLEAN WM_USER+1
#define APP_MOVE WM_USER+2

CONST NumEntries = 100;
```

The CONST keyword is available as an aid to converting code from other languages. A pre-processor definition defines an identifier as either true or false and can be used to control whether code is compiled or not.

```
#ifndef MYINCLUDE
#define MYINCLUDE
...code here
#endif
```

String literals

A string literal is text enclosed in quotes. The compiler supports string escape sequences to insert special ASCII values into the string, these special character are those that would be hard to type, or have special meaning such as the new-line character.
Consider the following examples:

```
Mystring = "This is a string";
// a string with embedded quotes
MyString2 = "This \"is\" a string";
```

All escape sequences begin with a single backslash ' \ '.
In order to have a backslash in the string itself you need to use a double backslash ' \\ '.
This is important to remember when working with filenames.

```
Myfile = "C:\\data\\inp.text";
```

Supported escape sequences:

Sequence	Inserted character
\n	Newline character
\t	TAB character
\\	A single backslash
\"	A quote
\xnn	An ASCII character whose hex value is 'nn'.

The last sequence deserves a bit of explanation. The standard ASCII character set is an integer number from 0 to 127. Or in hexadecimal notation 0x00 to 0x7F. The extended character set from 0x80 to 0xFF and the non printable characters from 0x00 to 0x31 are not normally available from a standard keyboard, or have special meanings to a text editor. The \x sequence allows embedding these characters into the string.

Example:

```
strOut = "\x1BThere is an <ESC> character at the beginning";
```

The maximum length of a string literal is 1023 bytes. If you need a longer string literal then append them together with the concatenation operator. String literals are stored directly in the executable file.

Duplicate string literals in a program will be combined into a single instance in the executable.
This can cause a side effect that if you assign a literal to a pointer and change that literal through dereferencing, all references to that literal will show the modified literal.
Always use a variable in this case.

Numeric literals

Numeric literals are sometimes called numeric constants. We use the term literals to differentiate them from the CONST keyword and to avoid confusion. A numeric literal is simply a number, either integer or floating point, entered directly into the source code.

Such as when assigning a value to a variable:

```
A = 1.3452; //The number is the literal
```

Direct entering of exponents is also allowed by the compiler.

```
A = 1.2e-10; //Set a to equal 0.00000000012
```

Hexadecimal numbers may also be directly specified by using the 0x identifier.

```
A = 0x500 + 0x2FFF;
```

Numeric modifiers

The compiler supports modifiers to the entered number to change its type.
If a number is entered without a decimal point it is treated as an INT type (4 bytes) , with a decimal point it is treated as a DOUBLE type (8 bytes).
It is sometimes desirable to modify the type of the numeric to tell the compiler how to convert and store it. For example an INT literal is not large enough to hold an INT64 literal.
Numeric modifiers appear as a single character following the literal.

```
//A will be defined as an INT64
//If not already defined
A = 36674965736284q;

//flNum will be defined as type FLOAT
//if not already defined
flNum = 1.234f;
```

Supported modifiers:

Modifier	Result
q	INT64
f	FLOAT
u	unsigned int

The u modifier was specifically designed for entering hexadecimal numbers as an unsigned quantity. Normally all non decimal numbers are treated as a signed integer which can cause calculation problems when using hexadecimal numbers.

```
A = 0xFFFFFFFFu -1u;
```

Without the u modifier A, if not already defined, would have a type of INT and a value of -2.
By specifying the u modifier, A is defined as an unsigned integer with a value of 0xFFFFFFFE or 4294967294.

All about loops

Now that we have the basics of variables and constants down we can now move on to loops.
A loop is just what it sounds like, a portion of code that gets repeated a number of times based on a condition. Aurora supports three basic loop types. The FOR loop, the WHILE loop and the DO loop.

FOR Loop

This loop will most likely be the most used construct in your program. Aurora's FOR loop is constructed by specifying three expressions. The initialize expression, the conditional expression and the loop expression.
The syntax of FOR looks like this:

```
for( init_expr; cond_expr; loop_expr)
  {
  statements;
  }
```

statement can be either a single program statement or multiple program statements enclosed in braces.
init_expr is executed once before the beginning of the loop.
cond_expr is executed before each loop begins. If it is TRUE then the loop continues, if FALSE the loop exits.
loop_expr is executed at the **end** of every loop.

It may sound complicated, but in reality the simplicity of it gives the FOR loop a lot of power.
Let's start a new program so we can experiment with the FOR loop.
Create a new source file and save it to disk as Tutorial1A.src or a name of your choosing.
Type in the following short program:

```
global sub main( )
  {
  int count;
  for( count = 0 ; count < 20; count = count + 1)
    {
    print(count);
    }
  print();
  print("Count now equals: ", count);
  while GetKey() = "";
  }
```

Compile and run the program to see the results.
The initialize expression of our FOR loop is ***count = 0*** which stores the value of zero in our variable before the loop begins.
The conditional expression is ***count < 20*** which compares our variable to the number 20 **before** each loop begins. If our variable is less than 20 then the loop continues.
Our loop expression is ***count = count + 1*** which will add 1 to our variable at the **end** of each loop iteration.

Creating a loop that counts backwards is a simple matter of changing our conditional expression and loop expression.

Change the following lines, in red, in your source code window:

```
global sub main( )
     {
     int count;
     for( count = 20 ; count > 0; count = count - 1)
       {
       print(count);
       }
     print();
     print("Count now equals: ", count);
     while GetKey() = "";
     }
```

Compile and execute the program to see the difference.

The FOR loop can use more than just numbers in its expressions. Any expression that you can test a condition on, can be used in the loop which makes it a very powerful tool.

Consider the following loop:

```
string s;
int x=0;
for( s = "A"; s[ x ] < "Z"; x++)
   {
   s[x+1] = 'B' + x;
   }
```

Can you figure out what the loop does without running it?

Replace the loop in our program with the one above, and add a *print(s);* statement.
Your program should look like this:

```
global sub main( )
   {
   string s;
   int x=0;
   for( s = "A"; s[ x ] < "Z"; x++)
     {
     s[x+1] = 'B' + x;
     }
     print(s);
     while GetKey() = "";
   }
```

WHILE Loop

Next on the agenda is the WHILE loop.
The WHILE loop repeats one or more statements while a condition is TRUE.
The condition is tested at the beginning of each loop.
To see how a WHILE loop operates let's change the program above. so:

```
global sub main( )
   {
   string s = "A";
   int x=0;
   while( s[ x ] < "Z" )
      {
      s[x+1] = 'B' + x;
      x++;
      }
   print(s);
   while GetKey() = "";
   }
```

As you can see, both the FOR and WHILE loops continue iterating while a condition is TRUE.

DO Loop

Last but not least is the DO loop.
The DO loop repeats one or more statements **until** a condition is true.
The condition is tested at the bottom of the loop so the statements will always execute at least once.

Change the program as follows to see a DO loop in action:

```
global sub main( )
   {
   string s = "A";
   int x=0;
   do
      {
      s[x+1] = 'B' + x;
      x++;
      } until( s[ x ] = "Z" );
   print(s);
   while GetKey() = "";
   }
```

Operators and mathematic expressions

An operator is a symbol that performs a specific function on a variable, constant or identifier. Operators consist of **mathematic** operators, **conditional** operators, **boolean** operators and **control** operators. Some symbols are reused for other purposes depending on the context of the operator.
For example the & symbol is used as the bit wise AND operator and also as the Address Of operator

The Aurora compiler understands the following operators:

Symbol	**Meaning**
+	Addition / String concatenation.
-	Subtraction / Unary minus.
*	Multiplication / Pointer dereferencing / Pointer definition.
/	Division.
%	Modulus. Remainder of an integer division.
^	Raise to a power.
=	Assignment / Equality.
==	Equality.
|	Bitwise OR.
||	Logical OR.
&	Bitwise AND / Address Of.
&&	Logical AND.
	Exclusive OR.
AND	Logical AND.
OR	Logical OR.
>>	Bit shift right.
<<	Bit shift left.
->	Pointer dereference and member access to struct class.
.	Member access to struct/class
>	Greater than.
<	Less Than.
>=	Greater than or equal.
<=	Less than or equal.
!=	Not equal.
< >	Not equal.
!	Boolean NOT.

The operators with the highest precedence level are evaluated first unless overridden with parenthesis. Operators are evaluated from left to right ,ordered by the following precedence rules.

Precedence level	**Operators**
1	&, |, AND, OR, &&, ||
2	>, <, >=, <=, !=, <>, =, ==, |=, &=, *=, /=, -=, +=
3	>>, <<
4	+, -
5	/, *, %, XOR
6	^
7	(,)
8	., ->

Conditional Statements

A conditional statement executes one or more program statements based on a condition.
These include IF, ELSE, and SELECT (SWITCH).

The IF statement

The IF statement will probably be your most used conditional statement.
The statement has two forms, the block IF and a single line IF.
The block form executes one or more statements if a condition is TRUE

Create a new source file and save it to disk as Tutorial1B.src or a name of your choosing.
Type in the following short program:

```
global sub main( )
   {
   int num;
   do
      {
      print("Enter a number - 999 to quit ",);
      num = StrToNum(readln());
      if ( num != 999)
         {
         print("Answer = ", num + 2);
         }
      } until num = 999;
   print("Press any key to close");
   while GetKey() = "";
   }
```

Aurora supports both the ELSE and ELSE IF keyword pairs.
When combined with an IF statement, the statement(s) following an ELSE will execute if the condition is FALSE. An ELSE IF statement will begin a new IF conditional when the previous condition was FALSE.
IF/ELSE statements may be nested.

For example:

```
global sub main( )
   {
   int num;
   do
      {
      print("Enter a number - 999 to quit ",);
      num = StrToNum(readln());
      if ( num != 999)
         {
         if(num < 100)
            print("Answer = ", num + 2)
         else if(num < 200)
            print("Answer = ", num * 2)
         else
            print("Answer = ", num / 2);
         }
      } until num = 999;
   print("Press any key to close");
   while GetKey() = "";
   }
```

As illustrated in the example above a single semicolon is used to terminate a group of single line IF and ELSE statements. If you use the block form, by placing one or more statements in braces, you must terminate each line with a semicolon.

```
global sub main( )
  {
  int num;
  do
    {
    print("Enter a number - 999 to quit ",);
    num = StrToNum(readln());
    if ( num != 999)
      {
      if(num < 100)
        {
        print("Answer = ", num + 2);
        }
      else if(num < 200)
        {
        print("Answer = ", num * 2);
        }
      else
        print("Answer = ", num / 2);

      }
    } until num = 999;
  print("Press any key to close");
  while GetKey() = "";
  }
```

The SELECT (SWITCH) statement

The SELECT statement is a conditional statement that allows you to test a single expression against one or more conditions. A distinct CASE label is used for each condition.
SWITCH is an alias for SELECT and is commonly used in the C language.

CASE statements can be grouped together by using the CASE& keyword. If none of the CASE comparisons are TRUE an optional DEFAULT statement can be specified as a catch all.

Each CASE, CASE& or DEFAULT statement must be terminated by a colon.

Type in the following short program to illustrate.:

```
global sub main( )
  {
  byte c;
  int done = 0;
  print("press some keys, Q to quit");
  do
    {
    do
      {
      c = GetKey();
      } until c != 0;
    select(c)
      {
      case 'A':
      case& 'a':
        print("You pressed \"A\"");
      case 'Z':
      case& 'z':
        print("You pressed \"Z\"");
      case 'Q':
      case& 'q':
        done = true;
      default:
        print("The ASCII value is ",c);

      }
    } until done;
  print("Press any key to close");
  while GetKey() = "";
  }
```

Writing Subroutines

Subroutines are sections of programs that need to be executed numerous times. Subroutines can be called from anywhere in the program and return execution from where they were called. Functions, commands, statements, API, procedure, and handlers are all common names for a subroutine. The distinction of the name depends on the usage.

All subroutines must begin with the SUB or GLOBAL SUB keywords. The statements of a subroutine are enclosed in braces **{ }**. The subroutine can appear anywhere in your source files and do not affect the path of execution until you call them. Subroutines do not need to be declared if they are only used in the source file they are defined in. You must declare a subroutine as external if you wish to use a subroutine located in another source file, this will be covered shortly.

In the SUB statement, list any parameters that will be passed to the subroutine and also whether or not your subroutine returns a value. Here is a short example subroutine

```
SUB MyFirstSub( int num ), int
   {
   INT temp;
   temp = num * 4;
   RETURN temp;
   }
```

The variables defined in your subroutine, or listed as a parameter in the SUB statement are called *local variables*. A local variable is only accessible while your subroutine is being executed. Once control is returned to the code that called the subroutine the local variable does not exist. All local variables are stored in a special area of memory called the stack.

The stack size is set by the 'Advanced' tab on the *Executable Options* dialog or the *Project Options* dialog. The default stack commit size is set to 32K and indicates how large the size of the local variables in any one subroutine can be. If you exceed this size the compiler will generate a warning message so you can adjust the stack size as needed.

A RETURN statement is only needed if you are returning a value or if you wish to return from your subroutine at a point in the code other than the end of the subroutine. The compiler automatically inserts the RETURN statement at the end. You will get a warning message if you forget a RETURN statement in a subroutine that returns a value.

Calling the subroutine

A subroutine is called, or jumped to, by simply using its name and specifying the parameter values to pass to the subroutine. If a subroutine does not have any parameters, you must still use an empty set of parenthesis ( ). A subroutine that returns a value can be used anywhere you would use that type of value.

Type in and run this example program to illustrate subroutines:

```
global sub main( )
  {
  Cls();
  CenterPrint("Subroutine Example",1);
  CenterPrint("Today Is: "+DateTime(),6);
  CenterPrint("Press any key to close",11);
  WaitForKey();
  }

SUB WaitForKey()
  {
  while GetKey() == "";
  }

SUB CenterPrint(string text,int row)
  {
  Locate(row, (80 - len(text)) / 2);
  print(text);
  }

SUB DateTime(),string
  {
  string strReturn;
  strReturn = FormatDate() + " " + FormatTime();
  return strReturn;
  }
```

Passing parameters

The above example shows passing values to a subroutine. Variables can be passed by either value or reference depending on the type of variable and whether or not the BYREF or BYVAL keyword is used. When a variable is passed by reference the subroutine is allowed to make changes to the contents of the passed variable. When a variable is passed by value, a copy of the contents of the variable is sent to the subroutine and the original variable will remain unchanged.

Aurora defaults to passing by value for all numeric types and passing by reference for strings, structures and arrays. The following table lists the defaults:

Type	**Default passed by**
BYTE	Value
WORD	Value
INT	Value
FLOAT	Value
DOUBLE	Value
INT64	Value
STRING	Reference
WSTRING	Reference
POINTER	Reference
Structure	Reference*
Array	Reference
Class/Interface	Reference

* Structures can be passed by value by using the BYVAL keyword.
Numeric types can be passed by reference using the BYREF keyword or using a pointer to type.

Passing Arrays

Arrays require special handling when passing them to a subroutine. The subroutine needs to know the dimensions of the array at compile time. For single dimensioned arrays it is not necessary to supply a dimension and an empty bracket set will suffice '[ ]', but it is good procatice to specify even the single dimenstion. All arrays are passed by reference so anything done to the array in the subroutine will modify the array passed to it.

This example illustrates passing arrays to a subroutine:

```
global sub main()
   {
   int myarray[10,50];
   float floatarray[10];

   ArraysAreFun(myarray, floatarray);
   for(int i=0;i< 10;i++)
      {
      for(int j=0;j<50;j++)
         {
         print(myArray[i,j],",",);
         }
      }
   print("\n");
   for(i=0;i<10;i++)
      {
      print(floatarray[i]," ",);
      }
   print();
   while GetKey() == "";
   }

SUB ArraysAreFun(int iArray[10,50], float flArray[10])
   {
   for(int i=0;i< 10;i++)
      {
      for(int j=0;j<50;j++)
         {
         iArray[i,j] = i*50+j;
         }
      }
   for(i=0;i<10;i++)
      {
      flArray[i] = fsind(i);
      }
   }
```

Optional Parameters

Optional parameters at the end of the parameter list may be specified using the OPT keyword. A default value may be specified and will be used if the parameter is not included in the calling list. If a default parameter is not included then either 0 or a NULL string will be passed to fill in the optional parameter.

```
global sub main()
   {
   string temp;
   print("Hello " + GetString("Enter Your Name: "));
   print("Press return to close");
```

```
        temp = GetString();
        }

    SUB GetString(opt string *prompt),string
        {
        if(prompt)
            print(*prompt,);
        return readln();
        }
```

Declaring subroutines

The DECLARE statement is used to tell the compiler about a subroutine that might be located in another file, in a DLL or using a different calling convention such as CDECL (C declaration). When a global subroutine is compiled in another source file you can use it in any other source file, in the same project by using the DECLARE EXTERN keywords.

A *global subroutine* is one that has been created with the GLOBAL SUB keyword pair. This allows the subroutine to be listed in a special table used by the linker. When the linker sees a call to an external subroutine it looks up the address of that subroutine in the table and resolves the reference to it. To really see global subroutines in action you need to create a project, add two or more source files, and create a few global functions to test.

```
    //file1.src
    declare extern SomeSub(int a),int;
    declare extern AnotherSub(string s),string;

    global sub main()
        {
        print(SomeSub(10));
        print(AnotherSub("Hello There"));
        print("Press any key to close"));
        while GetKey() == "";
        }

    ------------------------------

    //file2.src
    global SUB SomeSub(int a),int
        {
        return a * 2;
        }

    global SUB AnotherSub(string s),string;
        {
        return s + ", nice to meet you";
        }
```

All global subroutines must be uniquely named or linking your executable will fail with a "duplicate definition error". When debugging, only subroutines declared as global will display in the context window when a breakpoint is encountered. You will still get the correct file name and line number, just the name of the subroutine will default to the last global name in the path of execution.

You can also use the DECLARE statement for subroutines located in the same file, but it is not necessary with Aurora. When the DECLARE statement is used locally it will override the parameter names specified in the SUB statement, so be sure to name them the same to avoid confusion.

Calling conventions

A calling convention controls how the compiler calls a subroutine, how the parameters are pushed, and how return values are handled. By default Aurora uses the STDCALL calling convention which is standard in many other languages. As mentioned previously you can also use the CDECL calling convention which is standard with the C language. CDECL allows creating subroutines with a variable number of parameters. The 'sprintf' function in the C runtime library is a good example of a subroutine that takes a variable number of parameters and must be declared using the CDECL keyword before it can be used.

A short example:

```
DECLARE CDECL import,sprintf(out as STRING,format as STRING,...),INT;

global sub main()
  {
  string buffer;
  sprintf(buffer, "%s 0x%X\n", "The answer in hexidecimal is:", 32766);
  print(buffer);
  print("Press any key to close");
  while GetKey() == "";
  }
```

Creating a variable argument subroutine

The previous example illustrated how you would declare and call a subroutine that accepts a variable number of arguments. You can write a subroutine that accepts a variable number of arguments as well.

The ellipses (...) when used as the last parameter of a subroutine indicates to the compiler that it can accept any number of arguments. The subroutine must have at least one parameter before the ellipses. Aurora automatically uses the CDECL convention when it see the ellipses so specifying it in a declare statement is unnecessary.

```
SUB AddItUp(int num, ...)
```

To get to the variable argument list use the VA_START function with the parameter preceeding the ellipses. VA_START returns a pointer to the passed arguments on the stack and it must be the first statement in your subroutine.

```
void *args = VA_START(num);
```

The variable arguments are passed without regard to type and size information. It is up to the programmer to determine the size of each argument sent to the subroutine. A common methods is to use an index variable or formatting string. For example The USING function, and the C sprintf function, use a variable number or arguments whose type and size depends on a formatting string.

The size of a type can be determined using the LEN operator.

```
//get the first argument
arg1 = *(double)args;
args += len(DOUBLE);
//get the second argument
arg2 = *(int)args;
```

It is important to remember that how you access an argument is entirely up to your design. By default all numeric variable types are passed by value. To keep inline with industry standards all floating point types are converted to a DOUBLE before being passed. Strings and structures are passed by reference.

This example illustrates creating a subroutine that accepts a variable number of arguments:

```
global sub main()
  {
  //Call the variable argument subroutine
  //Add up 4 numbers
  print( "Sum of 20.5, 10.0, 40.32, 60.1" );
  print( AddItUp(4, 20.5,10.0,40.32,60.1),"\n" );
  //Add up 7 numbers
  print( "Sum of 1.32, 5.0, 1.0, 10.7, 100.74, 845.25, 721.11" );
  print( AddItUp(7, 1.32,5,1,10.7,100.74,845.25,721.11),"\n");

  //Wait until closed
  print("Press any key to end");
  while GetKey() == "";
  }

SUB AddItUp(int num,...),double
  {
  void *pArgs;
  double flTemp = 0.0f;
  //get a pointer to the first unknown argument
  pArgs = VA_START(num);
  //Add up all variable arguments
  for(int x=0;x < num; x++)
    {
    flTemp += *(double)pArgs;
    //A DOUBLE precision number is 8 bytes long
    pArgs += len(DOUBLE);
    }
  //return the sum
  RETURN flTemp;
  }
```

Function pointers

Aurora supports function pointers, which allows indirectly calling a subroutine through the use of a special pointer. The function pointer is created with the declare statement and can be assigned any address.

```
DECLARE *SomeFunction(int a, float b),int;
```

The name of the function is not important as it will be used to hold the address of a subroutine. The parameters must match the referenced function.

After a function pointer is initialized it can be used like any other subroutine. A common use for function pointers is dynamically loading a DLL and calling a function at run time. Using this method the case of a missing DLL file can be caught by your program and an appropriate error message, or return value, given.

Aurora uses this method in the database library:

```
DECLARE *SQLConfigDataSource(hwndParent as UINT,fRequest as WORD,lpszDriver as
STRING,lpszAttributes as STRING),INT;
CDatabase::CreateMDB(path as STRING),INT
  {
  def hlib as unsigned INT;
  def hfunction as unsigned INT;
  def szAttrib as STRING;
  INT rc = FALSE;
  ZeroMemory(szAttrib,255);
  szDriver = "Microsoft Access Driver (*.mdb)";
  szAttrib = "CREATE_DB=\""+path+"\" General\x0\x0";
  if(LEN(path))
    {
    hlib = LoadLibraryA("odbccp32.dll");
    if hlib
      {
      SQLConfigDataSource = GetProcAddress(hlib,"SQLConfigDataSource");
      if SQLConfigDataSource
        rc = SQLConfigDataSource(NULL,ODBC_ADD_DSN,szDriver,szAttrib);
      FreeLibrary(hlib);
      }
    }
  RETURN rc;
  }
```

Function pointers can also be used with your own subroutines. This example uses function pointers to create a four function calculator. You could do the same thing without function pointers of course, however, a comprehensive example is difficult to come up with in a small program.

```
// a function pointer
DECLARE *Operation(float p1, float p2),float;

global sub main()
  {
  //an array to hold subroutine addresses
  UINT fnArray[4];
  fnArray = &Addition,&Subtract,&Multiply,&Divide;
  //variable to hold input
  string calc;
  //variables to hold operands
```

```
    float p1,p2;
    //Operator index into the funciton array
    int op;

    Cls();
    print( "Simple Calculator" );
    print( "-----------------" );
    print( "Enter a calculation and press <ENTER>" );
    print( "Example: 24 + 23" );
    print( "Enter Q by itself to quit" );
    do
      {
      op = -1;
      print(">",);calc = readln();
      //first operand is easy, VAL stops at the first invalid character
      p1 = VAL(TrimLeft(calc));
      //go through the string to find the operator
      i = 0;
      while calc[i] && op = -1
        {
        select calc[i]
          {
          case "+":
            op = 0;
          case "-":
            op = 1;
          case "*":
            op = 2;
          case "/":
            op = 3;
          }
        i++;
        }
      //did we find a valid operator?
      if op = -1
        print( "Invalid operator, Try Again" )
      else
        {
        //we found something to calculate
        //The next operand is after the operator
        p2 = VAL(TrimLeft(StrMid(calc,i+1)));
        //check for division by 0
        if op = 3 && p2 = 0.0f
          print( "ERROR: Attempted division by 0" )
        else
          {
          //perform the calculation using an indirect function call
          Operation = fnArray[op];
          result = Operation(p1,p2);
          print( result );
          }
        }
      } until calc = "Q" || calc = "q";
    }


  // Define the subroutines to be called indirectly
  SUB Addition(float p1,float p2),float
    {
    return p1 + p2;
    }
```

```
SUB Subtract(float p1,float p2),float
   {
   return p1 - p2;
   }

SUB Multiply(float p1,float p2),float
   {
   return p1 * p2;
   }


SUB Divide(float p1,float p2),float
   {
   return p1 / p2;
   }
```

End of Part 1.

Part 2 - Continuing with Aurora

In Part 1 of this tutorial we covered basic program structure, console I/O, loops, conditional statements and writing subroutines. In Part 2 we are going to cover file operations, pointers, using external libraries and basic OOP usage.

Pointers, references, and type-casting...oh my!

Pointers are arguably the least understood concept new programmers face. They are also the number one source of errors you will encounter while programming. With that said you will find that they are impossible to live without.

A pointer is, simply stated, a variable that holds the memory address of another object. When we say object, we are not talking about a class object, although a pointer to a class is a common use, but instead anything in your program that exists in memory.

Aurora supports both ***typed pointers*** and ***un-typed pointers***. A typed pointer contains the address of the object and the compiler knows the type of the object it points to. An un-typed pointer is just an address and in order to use it a ***typecast*** must be specified.

Once a pointer has been assigned a memory address, or reference, the memory can be accessed using a ***dereference operator***. Aurora uses the * character as a dereference operator and the **->** symbol combination to access a member of a structure, class or to call a class or interface method when a pointer is given as a left hand operand.

Let's begin with a typed pointer example. A typed pointer is defined by using the * character:

```
int *memory;
```

The above statement is comprised of three parts:

int is the type of data,
* tells the compiler that it is a pointer, and
'**memory** is the name of the pointer variable.

The next step is to assign an address to the pointer variable. Aurora uses the **&** symbol to mean ***address of***. This is common in other languages, but not really a necessity since the compiler is smart enough to know what to do when you assign a variable to a pointer. For readability it should be used anyway.

```
int iValue;
memory = &iValue;
```

At this point the variable called 'memory' contains the address of the integer variable 'iValue'. Now that the pointer is initialized we can modify the contents of the variable ‘iValue’ indirectly through the pointer. The operation of dereferencing a pointer is commonly called ***indirection***.

```
*memory = 1;
```

The statement above embodies both the simplicity and power of pointers. What is happening is the compiler is taking the memory address that is stored in 'memory' and storing the value of 1 into that address. On the outset you might be thinking that it is a round about way of assigning a value to a variable, and you would be right. Think about it on a broader scale though, such as a subroutine modifying a variable whose address has been passed as a parameter, and you will begin to see how important the pointer can be. Passing a variable **BYREF** is essentially the same as passing a pointer to that variable, only the syntax differs.

This example demonstrates basic typed pointer usage:

```
global sub main()
  {
  int *memory;
  int iValue;

  //assign the pointer the address
  //of iValue
  memory = &iValue;

  //Change iValue through the pointer
  *memory = 1;

  //print iValue through dereferencing and directly
  print("*memory = ",*memory," iValue = ",iValue);

  //print the address stored in 'memory'
  print("'memory' contains the address: ",hex$(memory));

  //send the address of iValue to a subroutine
  //it will modify the contents
  PointerDemo(&iValue);
  print("'iValue' now contains: ",iValue);
  print("Press any key to close");
  while GetKey() == "";
  }


  sub PointerDemo(int *i)
  {
  *i = 99;
  }
```

Typecasting

Type-casting allows access to the data, a pointer is referencing, as a different type. Or in the case of an un-typed pointer, it is required to let the compiler know what type of data you are accessing. In Aurora you specify the typecast surrounded by parenthesis during a dereferencing operation.

```
*(type_of_data)pointer_name
```

As mentioned earlier a pointer can be typed or un-typed. A void pointer is one of the ways you can create an un-typed pointer in Aurora. If we were to modify our example above to use an un-typed pointer it would look like this:

```
global sub main()
  {
  void *memory;
  int iValue;

  //assign the pointer the address
  //of iValue
  memory = &iValue;

  //Change iValue through the pointer
  *(int)memory = 1;

  //print iValue through dereferencing and directly
  print("*memory = ",*(int)memory," iValue = ",iValue);

  //print the address stored in 'memory'
  print("'memory' contains the address: ",hex$(memory));

  //send the address of iValue to a subroutine
  //it will modify the contents
  PointerDemo(&iValue);
  print("'iValue' now contains: ",iValue);
  print("Press any key to close");
  while GetKey() == "";
  }

sub PointerDemo(void *i)
  {
  *(int)i = 99;
  }
```

As you can see, using an un-typed pointer requires telling the compiler what type of data we are referencing.
A void pointer is still a pointer, it just does not have a preset type. The real use of an un-typed pointer is to be able to access memory as any type of data. Which brings up another topic, the **NEW** operator. When you allocate memory using the **NEW** operator, you are given a chunk of memory to store whatever data your program needs.

The **NEW** operator takes a type and a count.

```
void *data = NEW(byte, 100);
```

Internally Aurora is multiplying the length of the type by the count. We could allocate the same amount of memory by doing:

```
void *data = NEW(INT, 25);
```

An integer is 4 bytes long * 25 = 100 bytes. The important thing to remember is that NEW just returns a pointer to an allocated block of memory of the requested size. The memory doesn't have a type and can be used for whatever reason you see fit. When you are done with the memory you must return it to the system by using the **DELETE** operator.

```
DELETE data;
```

With that information we can easily create dynamic arrays. Arrays are normally a fixed size as the compiler needs to know the size of the array at compile time. By using **NEW**, **DELETE** and a pointer, you can create a single dimensional array of any size. Using a typed pointer the only difference between accessing a fixed single dimensional array and a dynamic one is the * operator.

Example of a dynamic array:

```
global sub main()
    {
    //pointer to word sized data
    word *array;

    int size;
    print("Enter the size of the array > ",);
    size = StrToNum(readln());
    if(size <= 0)
    size = 1;
    array = new(word,size);

    //fill in the array with some random values
    for(int x = 0; x < size; x++)
       {
       *array[x] = rand(1000);
       }

    //show the contents of the array
    print("\nArray contains:");
    for(x = 0; x < size; x++)
       {
       if(x == size-1)
          print(*array[x])
       else
          print(*array[x],",",);
       }

    //delete the array
    delete array;
    print("\n\npress any key to close");
    while GetKey() == "";
    }
```

Pointer math

A pointer contains an address and that address is a 32 bit unsigned integer value. Since it is a value we can make the pointer look at a different address by using any math operator. This is known as ***pointer math*** and it is a very powerful tool. We know that a dynamic array of same sized data can be created and accessed using a typed pointer. Using typecasting and pointer math you can access the memory as different types of data.

As an example problem, let's say that you wanted to store some floating point data and wanted a header that contains the number of data items with a description. After the description we want a variable number of floating point numbers. It is common for programs to store binary data in this manner and this is one method of accessing that data.

```
global sub main()
     {
     //pointer to any sized data;
     void *mydata,*pData;
     int size;
     string dataname;
     print("Enter the number of data items > ",);
     size = StrToNum(readln());
     print("Enter the name of the data set > ",);
     dataname = readln();
     if(size < 0)
     size = 0;

     //our header is one integer and one string.
     mydata = new(byte, len(int) + 255 + len(float) * size);

     //use a second pointer for accessing the data
     pData = mydata;

     //store the number of data items;
     *(int)pData = size;

     //move the pointer to store the string
     pData += len(int);

     //store the data set name
     *(string)pData = dataname;

     //move the pointer to store the data
     pData += 255;

     //store some random numbers as the data items.
     for(int x=0;x < size;x++)
        {
        *(float)pData = rnd(1000.0);
        pData += len(float);
        }
     print("--------------------");

     //now we have a block of memory that has a header
     //containing the number of data items and a string description
     //followed by the data itself. Print it all out to verify.
     //reset the pointer
     pData = mydata;
     size = *(int)pData;
     print("Number of items: ",size);
```

```
        pData += len(int);
        dataname = *(string)pData;
        print("Dataset name: ",dataname);
        pData += 255;
        if(size)
           {
           print("Data Items:");
           for(x = 0; x < size; x++)
              {
              if(x == size-1)
                 print(*(float)pData)
              else
                 print(*(float)pData,",",);
              pData += len(float);
              }
           }

        //delete the array
        delete mydata;
        print("\n\npress any key to close");
        while GetKey() == "";
        }
```

The example used typecasting for all references. A simplification could have been done by using a pointer to a float and accessing the floating point data in a typed manner.

When using pointer math it is important to remember that adding or subtracting from a pointer is done without regard to the type of data it references. **That is all math is performed byte wise**.

Member access operator

We briefly mentioned the **->** operator in the previous section. This operator goes by many names depending on what language reference your reading. Arrow operator, member selection operator, pointer to member, element selection operator and member access operator are a few of the names we have seen. No matter what you call it the member access operator performs the same function in nearly every language you will encounter it in.

In Part 1 of this tutorial we touched on the subject of structures or **UDT's** as they are commonly called. When you have a variable of a structure you access the members of that structure using the dot operator. But what if you had a pointer to memory allocated with **NEW** and wanted to use a typed pointer to access it as a structure? That is where the **->** operator comes into play.

```
point *pt = NEW(point,1);
pt->x = 10;
pt->y = 20;
```

The **->** operator is performing two tasks for you. It is dereferencing the pointer and accessing a member of the structure using the equivalent of the dot operator. The above code can be written as:

```
point *pt = NEW(point,1);
*pt.x = 10;
*pt.y = 20;
```

Which will produce the exact same machine code when it is compiled. The only difference is in syntax and readability. The **->** operator only works on typed pointers and is the **preferred method** of member access in this case. If you are using an un-typed pointer, you must use a typecast and in as much use the second method of accessing.

```
void *pt = NEW(point,1);
*(point)pt.x = 10;
*(point)pt.y = 20;
```

The member access operator is also used to call **class** and **COM** interface methods. We will go into more details on that later on. For now, let's change our last example to create a memory area that has a header consisting of an integer and a string followed by a variable number of structure records:

```
struct dailyprice
     {
     float open;
     float high;
     float low;
     float close;
     int volume;
     dstring date[11];
     }

global sub main()
     {
     //pointer to any sized data;
     void *mydata,*pData;
     dailyprice *price;
     int size;
     string dataname;
     print("Enter the number of days > ",);
     size = StrToNum(readln());
     print("Enter the name of the data > ",);
     dataname = readln();
     if(size < 0)
        size = 0;

     //our header is one integer and one string.
     mydata = new(byte, len(int) + 255 + len(dailyprice) * size);

     //use a second pointer for accessing the data
     pData = mydata;

     //store the number of data items;
     *(int)pData = size;

     //move the pointer to store the string
     pData += len(int);

     //store the data set name
     *(string)pData = dataname;

     //move the pointer to store the data
     pData += 255;

     //store some random stock info as the data items.
     price = pData;
```

```
        //start with an arbetrary date
        int jd = DateToJD(4,1,2005);

        //make sure it starts on a monday through friday
        if((jd % 7) > 4 )
           jd+= 8 - (jd % 7);
           for(int x=0;x < size;x++)
              {
              price->open = rnd(100,200);
              price->high = rnd(200,225);
              price->low = rnd(75,100);
              price->close = rnd(100,200);
              price->volume = rand(1000,10000);
              price->date = JDToDate(jd);
              price += len(dailyprice);
              jd++;
              if((jd%7) == 5)
              jd+=2; //only trades monday through friday ;)
              }
        print("---------------------");

        //now we have a block of memory that has a header
        //containing the number of data items and a string description
        //followed by the data itself. Print it all out to verify.
        //reset the pointer
        pData = mydata;
        size = *(int)pData;
        print("Number of items: ",size);
        pData += len(int);
        dataname = *(string)pData;
        print("Dataset name: ",dataname);
        pData += 255;
        price = pData;
        if(size)
           {
           print("\ndaily data:");
           print("date\t\topen\thigh\tlow\tclose\tvolume");
           for(x = 0; x < size; x++)
              {
              print(price->date,"\t",price->open,"\t",price->high,"\t",price-
              >low,"\t",price->close,"\t",price->volume);
              price+=len(dailyprice);
              }
           }

        //delete the array
        delete mydata;
        print("\n\npress any key to close");
        while GetKey() == "";
        }

   sub DateToJD(int month,int day,int year),INT
        {
        int jd,a,m,y;
        a = (14 - month)/12;
        m = month + (12*a) - 3;
        y = year +4800-a;
        jd = day + ((153*m+2)/5) + (365*y) + (y/4) - (y/100) + (y/400) - 32045;
        return jd;
        }
```

```
sub JDToDate(int JD),STRING
     {
     int a = JD + 32044;
     int b = (4*a+3)/146097;
     int c = a - (b*146097)/4;
     int d = (4*c+3)/1461;
     int e = c - (1461*d)/4;
     int m = (5*e+2)/153;
     int day = e - (153*m+2)/5 + 1;
     int month = m + 3 - 12*(m/10);
     int year = b*100 + d - 4800 + m/10;
     return using("0##/0##/####",month,day,year);
     }
```

The example above is a bit longer than others as I wanted to use a few julian date routines to create arbitrary monday - friday date strings.

File Operations

Aurora handles file operations using simple functions to open, close, read and write to a file.
The **OPENFILE** function creates or opens a file and returns a handle that will be passed to other file functions to operate on that file.

When using the **OPENFILE** function you specify the full path to the file and what ***mode*** the function should use.

The available modes are:

MODE_READ	- The file is opened for reading. If the file does not exist the function will fail and return NULL.
MODE_WRITE	- The file is opened for writing. If the file does not exist the function will fail and return NULL.
MODE_CREATE	- Creates a new file. If the file already exists it will be truncated to 0 length.
MODE_APPEND	- Opens a file for writing and moves the file pointer to the end of the file. If the file does not exist it will be created.

The modes can be combined to allow any operation on a file you desire. For example, if you want to open an existing file to add data to the end of it and also want to be able to read the file, you would specify

MODE_READ | MODE_WRITE | MODE_APPEND.

This short example creates a file called 'test.file' in the directory the executable is located in.

```
global sub main()
    {
    unsigned int hFile;
    hFile = OpenFile(GetStartPath()+"test.file",MODE_CREATE | MODE_WRITE);
    if(hFile)
      {
      print("File was created");
      CloseFile(hFile);
      }
    else
      print("File could not be created");
    print("Press any key to close");
    while GetKey() == "";
    }
```

As indicated in the above example any successfully created or opened file must be closed with the **CLOSEFILE** function before your program exits. If your program leaves a file open it will remain locked until the next reboot of the system.

Reading and Writing to files

After a file has been opened in the required mode it can be written to with the **WRITE** function. **WRITE** takes a handle to the file, a variable to write, and the number of bytes to write.
For binary files use the length of the variable type.
For text files, convert the variable to a string using either the **USING** function or the **NUMTOSTR** function.

It is important to note that the only difference between a binary and text file is the addition of newline characters at the end of each line. As far as the system is concerned a file is comprised of a sequence of bytes, nothing more.

As an example we will write the numbers 1 to 10 in binary to the file named 'test.file' and read them back.

```
global sub main()
    {
    unsigned int hFile;
    hFile = OpenFile(GetStartPath()+"test.file",MODE_CREATE | MODE_WRITE);
    if(hFile)
      {
      print("File was created");
      for(int x = 1;x<11;x++)
        {
        Write(hFile,x,len(int));
        }
      CloseFile(hFile);
      //now open the file again and read what was written
      hFile = OpenFile(GetStartPath()+"test.file",MODE_READ);
      if(hFile)
        {
        int num;
        print("Data in file = ",);
        while(Read(hFile,num,len(int)) == 0)
          {
          print(num,",",);
          }
        print();
        CloseFile(hFile);
        }
      }
    else
      print("File could not be created");

    print("Press any key to close");
    while GetKey() == "";
    }
```

The **READ** function takes as its parameters, a handle to a file opened for reading, a variable to store the bytes read from the file, and the number of bytes to read. As indicated above **READ** returns 0 on success or 1 on failure.

Text files

When working with text files it is necessary to read each line of a file as a separate entity. If you were to use the READ function you would have to find each return character and separate the lines manually.

Fortunately Aurora provides the **READSTRING** function which will sequentially read each line of a text file, one at a time, into the string variable that you specify.

The **READSTRING** function takes a handle to the file, a string variable, and the maximum number of characters to read. It is important to specify the maximum length to avoid overwriting the string. The maximum length should be the size of the string - 1 to account for the NULL terminator.

This example creates a small text file, reads it back, and then displays the contents in the console.

```
global sub main()
    {
    unsigned int hFile;
    string line;
    hFile = OpenFile(GetStartPath()+"test.file",MODE_CREATE | MODE_WRITE);
    if(hFile)
      {
      print("File was created");
      for(int x = 1;x<11;x++)
        {
        line = "This is line #" + NumToStr(x) + "\n";
        Write(hFile,line,len(line));
        }
      CloseFile(hFile);
      //now open the file again and read what was written
      hFile = OpenFile(GetStartPath()+"test.file",MODE_READ);
      if(hFile)
        {
        print("Contents of file:");
        while(ReadString(hFile,line,254))
          {
          print(line);
          }
        print();
        CloseFile(hFile);
        }
      }
    else
      print("File could not be created");

    print("Press any key to close");
    while GetKey() == "";
    }
```

The above example shows that when writing a string to a file to be used as text that you must add the newline character to the string.