x2c
xml-to-code

interpreter and language

License Agreement

This is a legal agreement between you and Cactus Computing bvba. By using this software, you are agreeing to be bound by the terms of this agreement.

Copyright. The x2c language and the code to implement it together with the documentation are owned by Man in The Middle AB and is protected by international copyright laws and international treaty provisions.

Grant of license. You are granted the right to install this software on more than one computer as long as it is only used by one user on one computer at a time. You are also granted the right to make copies of the software for backup purposes, but you may not under any circumstances give away or sell copies of the software to third parties.

No warranty. Man in The Middle AB disclaims all warranties, either express or implied, including but not limited to implied warranties or merchantability and fitness for a particular purpose, with respect to the software and the accompanying written materials.

Consequential damages. In no event shall Man in The Middle AB or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use this software product, even if Man in The Middle AB has been advised of the possibility of such damages.

This warranty gives you specific legal rights, and you may have other legal rights that vary by jurisdiction.

Table of contents

1 Introduction

XML is used more and more both to transfer data and to describe its structure. Often, we need to convert structures and data described in XML to other XML structures and then we use XSLT templates and apply them to the XML using some tool or browser. Sometimes, though, we want to transform XML data and data descriptions to something that isn't XML, like a straight text file, program source code or whatever. This can still be done with XSLT templates, but it quickly gets very difficult to get it right. XSLT templates aren't intended for procedural use, such as writing text files from top to bottom and has to be bent and forced a bit to allow it.

When producing program code things tend to be even more procedural than this (if something can be "more" procedural… you know what I mean). It looks more like standard code in text form with fill-in-the-blanks, where the data to fill in originates in XML files. This is even more difficult to do using only XSLT.

After writing some code generating stuff in C++, hard coding the code to generate into the programs, I increasingly felt the need to pull out the text templates from my code into some kind of templates so I didn't have to rewrite my program for other XML structures and to produce other source code files. The x2c interpreter and language is what finally came out of this effort.

The x2c interpreter is given a template file as command line parameter and then goes on to execute the script in that file. The script tells it where all the produced output should go and uses one or several XML files to look up data. The main output is written in the script file as fill-in-the-blanks text templates, making it very easy to see what you're doing. It's also making it easy to cut-and-paste text examples into the script and replace certain parts of that text with variables.

The x2c language is also provided with enough functionality to scan for files in directories, process any number of XML files, create new XML files, do some string processing, creating and applying XSLT transforms and more. It's a simple language, but extensive enough for most applications.

1.1 Limited free version

A limited free version of this product is available for download from www.wehlou.com/x2c. The free version has the following limitations:

- It is only for non-commercial use.
- Scripts are limited to maximum 200 lines, main and included files total, or 8000 characters, whichever comes first.
- Directory sets (see `dir()` function) limited to the ten first files found.

1.2 Installation

The x2c program is delivered as a self-decompressing archive which contains:

- x2c.exe: the interpreter executable, which is not dependent on external program modules or runtimes.
- docs folder: with this documentation file

- sample folder: contains one or more subfolders with sample scripts and files.

There is no need for a special installation or setup program, since x2c needs no settings from the environment or in the registry. Simply create a program folder like, for instance, c:\program files\cactus\x2c and copy the distribution file there. Then execute the distribution file from windows explorer by doubleclicking it, or from the command prompt by entering its name.
Since the x2c interpreter uses the MSXML engine, it will only work on windows machines with a recent Internet Explorer version installed.

1.3 Version history

In the below list, you'll find "alpha", "beta" and "final" versions. My understanding of these terms is as follows:

Alpha version. Not feature-complete. If features turn out to be lacking, they can still be added. Features may even change or disappear. When testing an alpha version, it's very sensible to let the author know what else you'd like to see in the product. Chances are good you may get your wish. The idea of alpha versions is to determine the feature set in cooperation with real and potential users, and relatively less attention is paid to fixing bugs.

Beta version. A beta version is supposed to be feature complete. That is, no features should be added or removed. Bugs should be hunted and killed with diligence.

Final version. A final version is supposed to be free of bugs. You wish… In reality, it's simply deemed good enough to use in production situations. Bugs will still be fixed, of course, except I'll be more ashamed of them than if they had been found in a beta version.

At any time and for any version, you are always welcome to report bugs and ask for features, of course. I'll do my best to track your reports and act on them at the appropriate time.

Release history

- **2001-07-30** V 0.1, first public alpha release of free version
- 2008-09-20 V1.1, implemented codepage based output

2 Program structure

An x2c program consists of a script file with the name of this file passed on the commandline. The script has a global (main) program area followed by any number of user defined functions. A very minimal script file could look like this:

```
// example of minimal script file
open create "min.out";
myfunc();

function myfunc()
{
      .A little output text
}
```

The script is automatically executed from the first available line and ends as soon as a return statement is encountered, a function definition is encountered or the end of the script file is hit.

The `#include` preprocessor statement allows you to split scripts into more manageable parts and to reuse functions and data across scripts. If you keep your functions separate, the above script could have looked like this:

```
// example of minimal script file
open create "min.out";
myfunc();

#include "myfunc.x2c";
```

Admittedly, it looks screwy not to have the `#include` statement at the top of the file. But since the `#include` does a straight insertion of the included file, that's the way it has to be.

2.1 Scoping rules

The x2c language implements scoping for variables and function. The global scope always exists and is accessible from any part of your script. Local scopes are added depending on blocks defined in your code.

In the global scope you'll find all variables declared in the global part of the script and the names of all functions defined by the script or predefined by the language.

Local scopes are brought into existence at several points in a script:

- Each user defined function lives in a local scope
- Each foreach construct has a local scope
- Each block (within curly braces) has its own local scope

Scopes are nested, so any inner local scope can access variables from outer local scopes and the global scope. Any locally defined variables with the same name as a variable in an outer scope overrides the outer variable, effectively hiding it from

access. As the innermost local variable goes out of scope, the outer variable again becomes accessible.

User defined functions are always defined at the global scope and are thus always accessible. Nested function definitions are not allowed.

2.2 Error handling

x2c is an interpreter, so the script code is only checked for errors as it is executed. A very few errors may be detected before the execution during the script loading and user-defined function scan. If an error is encountered, a reasonable error message is emitted and the program exits.

3 Data types

x2c has a minimal set of data types to handle most situations. Naturally, it has strings and numbers. All numbers are floating point. Booleans are provided. Since we're into XML processing, nodes and lists of nodes have their types too. Sets of strings are very useful for lists of files and lists of database entities, for instance. Maps (akin to dictionaries) are uniquely useful when you're doing code generation. Translating between concepts in XML into the corresponding concept in C++, Java, VB or any other language is the main use of these maps.

Conversion between data types is automatic. Whenever a string is expected, the actual data is converted to a string, for instance. Or whenever a number is expected and something else is passed, the appropriate conversion occurs. Since such conversions may be needed inside expressions, the operators in that expression must make clear what data type is expected and must be used. If the operators were the same for numerical addition and string concatenation, the following expression would be ambiguous:

```
"15" + "30"
```

It could evaluate to either "45" or "1530" depending on how you interpret it. The solution here is to use different operators:

`"15" + "30"` evaluates to the number `45`
`"15" & "30"` evaluates to the string `"1530"`

3.1 string

The string data type holds strings (duh).

Length

The maximum length of strings is for all practical purposes unbounded.

Constants

Strings can be delineated using either single or double quotes. If single quotes are used, double quotes are allowed as part of the string contents and vice versa.

Declaring strings

Strings can be declared as for example:

```
string s;
string s = "abc";
string s = "abc" & ucase("def");
```

Operations on strings

The only operation that can be carried out on string variables is concatenation using the '&' and '&=' concatenation operators.

```
string s = "abc" & "def";
s &= "ghi";
```

Comparisons

Lexical comparisons can be performed using the normal Boolean operators.
Examples:

```
if (s1 == "Hello") …
if (s1 <= s2) …
bool b1 = (s1 != s2);
bool bIsNotEmpty = s1;
```

Conversions

When a string is used in an arithmetic expression, its contents are parsed as a floating point number and converted. If the contents of the string cannot be seen as a number, the value "0" is used. Note that it is sufficient that the string begins with a number to have a conversion. Trailing characters are disregarded.
When a string is used in a Boolean expression, a non-empty string converts to "true", while an empty string converts to "false".

Functions

A number of built-in functions are available to convert between upper and lower case, to look up replacement strings in a map and more.

3.2 double

Defines a floating point numeric variable.

Range

Any double can be 1.7E +/- 308 (15 digits). That's what we call a decent range.

Declaring doubles

```
double d;
double pi = 3.14159;
```

A double declared without an initial value is initialized to 0.0.

Operations

All the common numerical operations can be performed:

```
a + b
a - b
a * b
a / b
```

A set of abbreviated operations without use of temporaries are also provided:

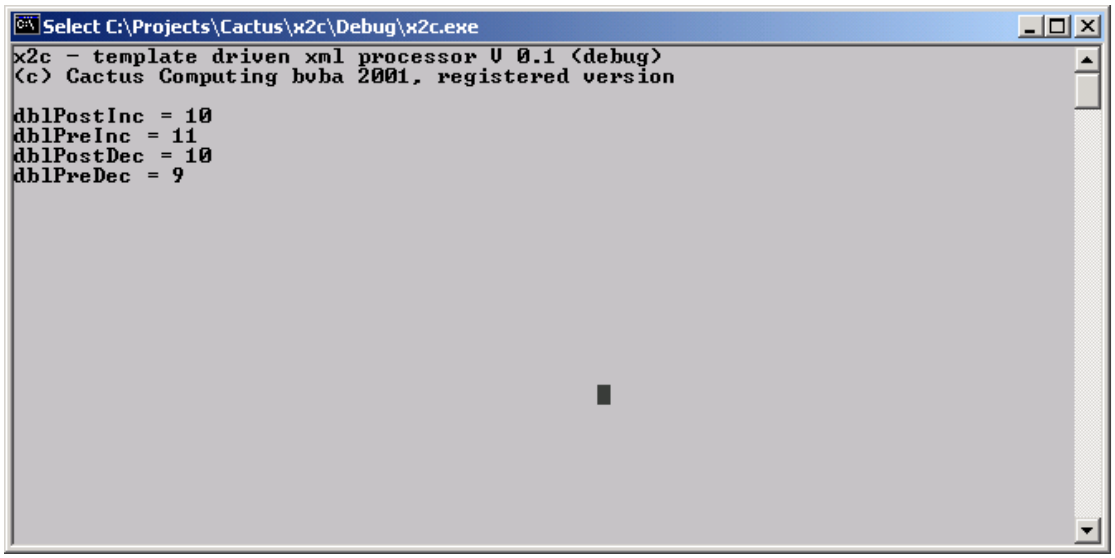
```
a += b     // same result as a = a + b
a -= b     // same result as a = a - b
a *= b     // same result as a = a * b
a /= b     // same result as a = a / b
```

Unary operators

There are unary operators provided for incrementing and decrementing numbers by one. Each of these unary operators can be applied before or after evaluation of the number a variable refers two. These operators can only be applied to variable names, not to expressions or constants.

```
double d;
d = 10; double dblPostInc = d++;
d = 10; double dblPreInc = ++d;
d = 10; double dblPostDec = d--;
d = 10; double dblPreDec = --d;
message("dblPostInc = " & dblPostInc);
message("dblPreInc = " & dblPreInc);
message("dblPostDec = " & dblPostDec);
message("dblPreDec = " & dblPreDec);
```

results in the following output:



What happens here is that in all cases d will be incremented or decremented by one, but if the operator is put before the variable, the increment or decrement will be performed *before* the value is used in the expression. Else the old value is used in the expression and only after that is the variable incremented resp. decremented.

Comparisons

Numbers can be compared the expected and usual way. The result of the comparison is treated as a Boolean value. Please take care to avoid problems caused by floating point round off errors.

```
a == b     // true if a equals b
a != b     // true if a does not equal b
a < b      // true if a is less than b
a > b      // true if a is greater than b
a <= b     // true if a is less than or equal to b
a >= b     // true if a is greater than or equal to b
```

3.3 bool

Defines a variable that can be true or false. These variables can be assigned to from expressions or constants.

Boolean constants

```
bool b1 = true;
bool b2 = false;
```

Expressions

```
bool b1 = "hi";  // evaluates to true
bool b2 = "";    // evaluates to false
bool b3 = 2 * 3;// non-zero evaluates to true, zero to false
bool b4 = (6 == 2 * 3);
```

Operators

```
bool b1 = b2 && b3;   // boolean AND
bool b1 = b2 || b3;   // boolean OR
bool b1 = !b2;        // boolean NOT
```

3.4 node

Defines a variable that can reference a node in an xml tree. That node can be the root node or any node below it. As long as any variable references any node in an xml tree, that xml tree will remain in memory.

Declaring, initializing and assigning

```
node n;
node n = node(xf, "object");
```

Node variables either hold a reference to a node or hold no reference at all. Assigning one node to another does not copy the contents of the node, only creates a second reference to the same node. For example:

```
node n = node(xf, "object");
node p = n;
settext(p, "help!");
string sText = gettext(n);
```

`sText` in this example will also contain the text "help!" since the variable `p` is pointing to the same node as the variable `n`.

3.5 nodelist

Defines a list of nodes as it is retrieved using an xpath expression in the *nodes()* function.

Example

```
nodelist nl = nodes(xf, "object");
```

The nodelist `nl` in the example will contain all the "object" elements under the parent xf. The list can be used in a `foreach` statement:

```
foreach node n in nl
{
     message("object name= " & evalattr(n, "name"));
}
```

3.6 set

A set contains any number of unique strings. Attempting to add a string to a set when it's already in the set does not change the set or cause an error.

Declaring and initializing

```
set s;
set s = {"abc", "def", "ghi"};
```

Adding / removing

```
set s2 = s1 + "jkl";  // s2 contains all of s1 plus "jkl"
s2 += "jkl";          // add element "jkl" to s2
set s2 = s1 - "jkl";  // s2 contains all of s1, except "jkl"
s2 -= "jkl";          // remove "jkl" from s2
s2 = null;       // empty the set
```

Union of sets

All elements present in one or both of two sets can be collected into a third set, effectively forming the union of the two sets:

```
set s1 = {"abe", "bob"};
set s2 = {"bob", "charlie"};
set s3 = s1 + s2;
```

The result is independent of the order the sets are added, so the last line above could just as well have been written:

```
set s3 = s2 + s1;
```

In both cases, the set s3 will contain the elements:

```
{"abe", "bob", "charlie"}
```

Intersection of sets

The intersection of two sets is the set of all strings that occur in *both* the original sets. You can form such a set by calling the built-in function `intersect()`:

```
set s1 = {"abe", "bob"};
set s2 = {"bob", "charlie"};
set s3 = intersect(s1,s2);
```

After the call, set `s3` contains only "bob".

Complement of sets

You can form a complement of one of the sets with respect to the union of the sets using subtraction. This sounds complicated, but it's actually nothing else than determining which elements are in one set but not the other. So:

```
set s1 = {"abe", "bob"};
set s2 = {"bob", "charlie"};
set s3 = s2 - s1;
set s4 = s1 - s2;
```

After this, set s3 will contain only "charlie", while s4 will contain only "abe".

Tests

```
bool bFound = find(s3, "bob");
bool bEqual = (s3 == s4);
bool bNotEqual = (s3 != s4);
```

Other comparisons (<=, >=, <, >) are not defined for sets.

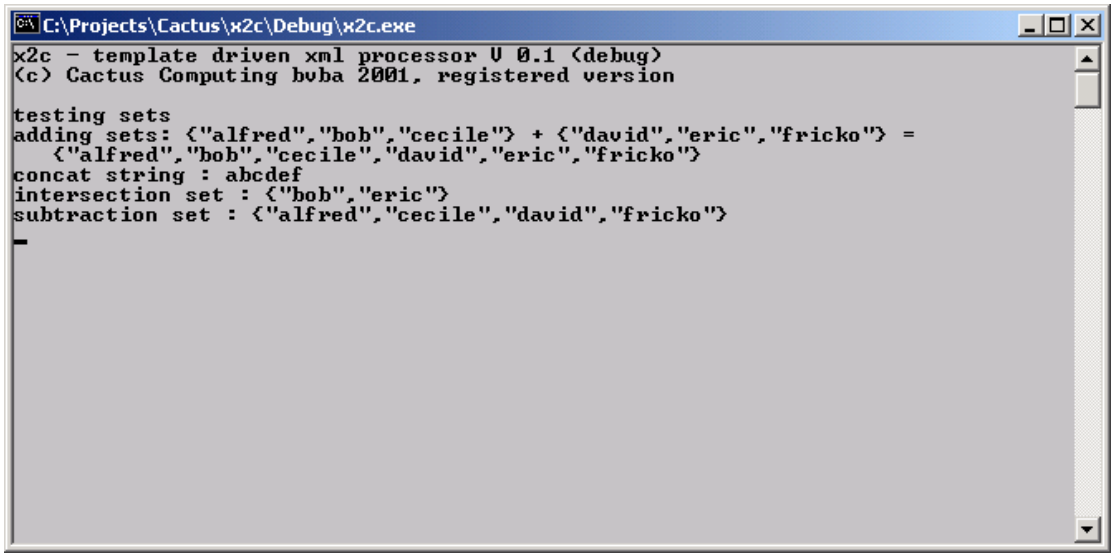
Examples

Think through the following test script and the adjoining output to see a few uses of sets:

```
// setsmaps.x2c
// testing sets and maps

message ("");
message ("testing sets");
set s1 = {"alfred", "bob", "cecile"};
set s2 = {"david", "eric", "fricko"};
set s3 = s1 + s2;
message ("adding sets: " & s1 & " + " & s2 & " = ");
message ("   " & s3);
set s4 = {"bob","charlie", "eric", "oscar"};

set s5 = intersect(s1 + s2, s4);
message ("intersection set : " & s5);
message ("subtraction set : " & ((s1 + s2) - s5));
```



3.7 map

A map contains key/value string pairs. There is no practical limit to how many such key/value pairs can be added to a map.

Declaring and initializing

```
map m;
map m = {"I" : "ik", "you" : "jij"};
map m = {"I" : "ik", "you" : "jij", default : error };
map m = {"I" : "ik", "you" : "jij", default : "not english" };
map m = {"I" : "ik", "you" : "jij", default : copyinput };
```

Any number of key/value pairs can be declared and a single optional default clause can be added. If the key isn't found during an unmap call (see below), the default clause determines the action to be performed. In case 3, a runtime error will occur. In case 4, the unmap call will return the string "not english". In case 5, the unmap call will return the input key itself unchanged.

Adding / removing

```
map m1 = {"I" : "ik", "you" : "jij"};
map m2 = {"he" : "hij"};
map m3 = m1 + m2;
m3 += m2;
map m3 = null;
```

Maps can be added to or merged, but not deleted from. If duplicate keys occur during a merge, a runtime error is triggered.

Resolving

```
string s = unmap(m3, "he");
```

Using the map definitions from the previous paragraph, the string s will get the contents "hij". Often, map values are resolved on the fly during output of template code like so:

```
.The word "he" translates to: "<$unmap(m3, "he")>" normally.
```

emitting the output:

```
The word "he" translates to: "hij" normally.
```

4 Keywords

4.1 Preprocessor

Preprocessor keywords are only interpreted once during the initial reading of the input files.

4.1.1 #include

Syntax

```
#include "filename.ext"
```

Description

The given file is inserted into the source file in its entirety as if the contents had been written into the source file at the location of the #include statement.
#include statements can be nested to any depth, but you must take care not to cause recursion by directly or indirectly including a file from other files included by it.
The #include keyword must be the first non-blank keyword in a line. The filename must be included in double quotes and anything following the closing quote, up to the end of the line, is ignored.

4.2 Runtime keywords

The following keywords are executed during runtime.

4.2.1 if

Syntax

```
if (expression) statement;
if (expression) statement else statement;
```

Description

This construct allows conditional execution of code. The `expression` can be any kind of expression and will be evaluated to a Boolean value as follows:

double:	non-zero values are "true"
string:	non-empty strings are "true"
node:	is "true" if the variable holds a node
nodelist:	is "true" if non-empty
set:	"true" if non-empty
map:	"true" if non-empty

Boolean variables and comparisons need no conversion, since they're Boolean to start with.

The `statement` part can be either a single statement or a block of statements within curly braces.

Example

```
if (name == "Alfred")
{
```

```
      name += " is dead";
      bAdded = true;
}
else
      name += "... who is this?";
```

4.2.2 foreach

Syntax

```
foreach string s in stringset statement;
foreach node n in nodelist statement;
```

Or:

```
string s;
foreach s in stringset statement;
```

```
node n;
foreach n in nodelist statement;
```

Description

The `foreach` construct allows repeating actions over a set of strings, a set of filenames or a set of nodes in a nodelist, for instance. The *`statement`* part will usually be a statement block within curly braces and seldom a single statement. The string or node declaration within the `foreach` construct will be in a local scope and will not exist beyond the `foreach` construct. The data type used in the variable immediately following the `foreach` keyword determines what kind of set or list the construct expects after the `in` keyword.

Example

```
node xf = xmlopen("myinput.xml");
nodelist nl = nodes(xf, "object");
foreach node n in nl
{
      string sName = evalattr(n, "name");
      message("object element attribute name: " & sName);
}
```

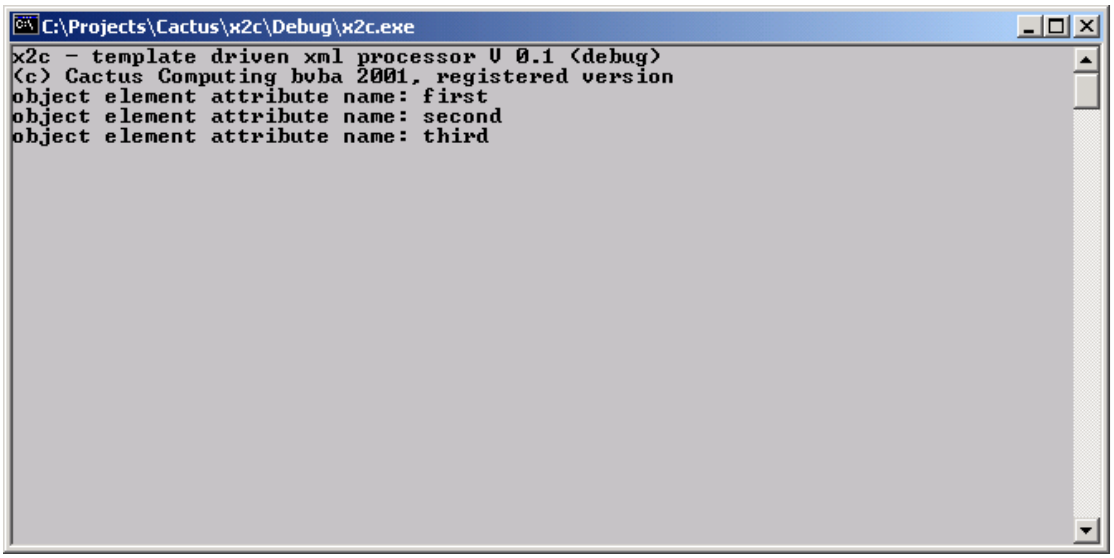
In this example, the xml file with the name "myinput.xml" is opened. The root element of the xml tree is returned and kept in the variable xf. All child elements with the tag "object" are selected and added to the nodelist nl. The foreach loop runs through these nodes and for each of them, prints the contents of the attribute "name" to the console.

If the xml file has the following content:

```
<?xml version='1.0'?>
<root>
      <object name='first'/>
      <object name='second'/>
      <object name='third'/>
```

```
</root>
```

…the console output will be:



Another very useful example is running through all xml files in a directory, as follows:

```
set sFiles = dir('*.xml');
foreach string sFileName in sFiles
{
      node xf = xmlopen(sFileName);
      ... do your thing here...
}
```

4.2.3 while

Syntax

```
while (expr) statement;
```

Description

Repeats the statement as long as the expression expr evaluates to true. The statement will often be a block of statements within curly braces

Example

```
double x = 20;
while (x > 10)
{
      message("x = " & x);
      x--;
}
```

4.2.4 break

Syntax

```
break;
```

Description

Use the `break` statement to terminate a `while` or `foreach` statement. If no surrounding `while` or `foreach` statement can be found, a runtime error occurs.

Example

```
double x = 20;
while(x > 10)
{
      if (x == 15) break;
      x--;
}
```

4.2.5 return

Syntax

```
return;
return value;
```

Description

Used to exit from user defined functions. If used in main program (global scope), it terminates the script processing. If returning from a user-defined function, any data type can be used as returned value. If returning from the main script, the return value will be converted to a number between 0 and 255 before returning only numbers or strings can be used as return value. If the value cannot be converted to a number in that range, the main program returns the value 0 to the operating system when terminating.

Example

```
function myfunc(double d)
{
      return d * d;
}
```

Returns the square of the input value.

4.2.6 open

Syntax

```
open <mode> filename [as (unicode|codepage <nb>)];

<mode>: create | append
<nb>: a codepage number installed on the system
```

Description

The `open` command redirects any subsequent text output to the named file. If any text output is open when the open command is encountered, that output file is closed first.

The first form (`open create`) erases any pre-existing file with the same name, while the second form (`open append`) positions itself to the end of the file, if a file can be found with the given name. If none exists, a new file is created in both cases.

The `open create` form is most useful for writing source code files, while the `open append` form allows adding statements to a make file or IDL file or lines to a log file, for instance.

The file is by default written in unicode, little-endian, starting with the 0xFEFF byte sequence that is standard for these files. By using the "as codepage <nb>" postfix, the user may define an MBCS output using any desired codepage. The codepage must be installed on the system running the script, else a runtime error will occur.

Example

```
set sInputFiles = dir('*.xml');
foreach string sInputFile in sInputFiles
{
      string sOutputFile = filename(sInputFile) & '.h';
      string sDefine = ucase(filename(sInputFile)) & '_H';
      open create sOutputFile as codepage 1251;
      .// Input file name: <$sInputFile>
      .#ifndef <$sDefine>
      .#define <$sDefine>
      … do your thing here…
      .#endif
}
```

In this example, we run through any number of xml files in the current directory and for each of these create an output file with the same name but with the extension '.h' instead. The first line in each of those files contain a comment with the name of the input file and a define in uppercase dependent on the output file name. We output the results of the script in ANSI Cyrillic, which is what codepage 1251 refers to.

4.2.7 <$…> construct

Syntax

```
<$expression>
```

Description

The <$…> construct may only appear within text output and marks an expression that is to be evaluated and replaced before the text is sent to the output file. Any results from the expression is converted to string form before being put into the output text. The <$…> constructs can be nested. See "text output operators" for more details and examples of use.

4.2.8 text output operators

Syntax

There are two distinct forms of the text output operators:

```
.This goes to the output
```

and:

```
[[ This goes to the output ]]
```

Description

The first form is line-oriented. It always outputs entire lines of text and appends a new line to the end of the line. The second form only includes new lines if they are present between the start and end markers of the output. So this second form allows inserting parts of lines. It is also handy for large blocks of text where you don't want to add a leading dot to every line.

Both text output operators output text to the current output file, which is determined by the most recently encountered `open` statement. If no `open` statement has been executed and thus no output file is available, a runtime error is triggered.

Everything after the dot in the first form, or between the brackets in the second form is first scanned for inline evaluations and then output to the current output file. Inline evaluations are performed wherever the `<$…>` construct is found without the output text. The `<$…>` constructs may be nested and the evaluations are performed from right to left.

Example

```
string sName = "Alfred";
string sDesc = "A person called";
[[So he said: '<$sDesc> <$sName>'...]]
```

results in the output:

```
So he said: 'A person called Alfred'...
```

Nested evaluation is done from right to left, so in the following example, the map key value will be evaluated before the `unmap` function gets called:

```
.His translated name is: <$unmap(namemap,<$sName>)>
```

Note that in this example as in most (but not all!) real world examples, the nesting wasn't really necessary. Examples were it is necessary are generally rather convoluted and not called for in a manual as this one. The following line would in fact result in exactly the same final output text:

```
.His translated name is: <$unmap(namemap, sName)>
```

See also

```
open create, open append
```

4.3 Userdefined functions

4.3.1 Defining functions

Syntax

```
function myfunc(parameterlist) : returntype
{
      function body
}
```

The *parameter list* contains any number and mix of data declarations (see *data types*). The *return type* is one of the built-in data types of the x2c language. If a return type is declared, all `return` statements in the function must return a value. If a return type is not declared, no `return` statement may return a value.

Example

```
function myfunc(double d, string s, node n) : string
{
      if (d == evalattr(n, s))
            return "right on";
      else
            return "no way";
}
```

This function may be used like:

```
node xf = xmlopen("myxml.xml");
node objnode = node(xf, "object");
.Is value really 2.0? ... <$myfunc(2.0, "value", objnode)>
```

4.3.2 Calling functions

Any user defined or built-in function is called by using its name and passing parameters within parenthesis. Sort of what you'd expect from a language.

Example

```
string s = myfunc(5, 7 * 3);

function myfunc(double d1, double d2) : double
{
      return d1 + d2;
}
```

Before the call, the expression `7 * 3` will be evaluated, so the function is called with d1 equal to 5 and d2 equal to 21. myfunc then returns the double value 26, which is converted to the string "26" before being assigned to the string s.

Calling built-in functions is entirely similar:

```
string s = ucase("help!");
```

…resulting in s containing "HELP!".

Since using the return value is optional, you may call a value returning function without assigning the return value:

```
ucase("I'm useless!");
```

Even though the above example is legal, it's also totally useless.

4.3.3 Returning from functions

The process returns from a function when a `return` statement is encountered, the beginning of a function definition is encountered or the end of the script file is hit.

A return statement may or may not refer to a variable or constant. If the function declaration specifies a return type, a value must be returned from the function. The caller is allowed to ignore returned values.

5 Built-in functions

5.1 XML related

5.1.1 xmlnew

Syntax

```
node = xmlnew(string sRootName);
```

Description

Creates a new, empty, xml DOM tree and gives the root element the name in the parameter. Unless the newly created xml tree is saved to disk using `xmlsave`, its contents will be lost when the variable goes out of scope.

5.1.2 xmlopen

Syntax

```
node = xmlopen(string sFileName);
```

Description

Opens the named xml file and returns the root element node. As long as the root element is in scope, or any variable holds any node of the document, the document still exists in memory. If the file is not found, a runtime error occurs. There is no explicit "close" function for xml files.

Note that if you do not supply a path with the filename, the system will look in the directory where the x2c executable is found, and in the paths supplied with the PATH environment variable setting.

5.1.3 xmlsaveas

Syntax

```
xmlsaveas(node root, string sFileName);
```

Description

If the given node is the root, the entire xml document is saved to the given filename. If the given node is a node within the xml DOM tree, that part of the tree is used to create a new DOM tree which is then saved. Note that if you want the file to be saved to the default directory, you should use the fully qualified name like so:

```
xmlsaveas(root, pathconcat(getcurdir() & "myfilename.xml"));
```

The xml tree can be a tree opened from a disk file using `xmlopen` or a tree created from scratch using `xmlnew`.

Note that if you supply a filename without a path, the file will be saved into the directory where the x2c executable file is found.

5.1.4 Evalattr

Syntax

```
evalattr(node parent, string attrname);
```

Description

Retrieves the textual value of the attribute named in the second parameter. The first parameter holds the node to find the attribute in. If an attribute with the given name is not found, an empty string is returned.

Example

An element looking like:

```
<object name='Alfred' scope='private'/>
```

Selecting this object and retrieving the value of the name attribute:

```
node nObj = node(parent, 'object');
string sNameVal = evalattr(nObj, 'name');
```

5.1.5 setattr

Syntax

```
setattr(node parent, string sName, string sValue);
setattr(node parent, string sName, double dblValue);
```

Description

Creates or replaces an named attribute of the parent node with the value given as string or double.

Example

```
node xr = xmlnew("rooty");
setattr(xr, "scope", "public");
```

Will result in a little xml tree looking like:

```
<?xml version='1.0'?>
<rooty scope='public'>
</rooty>
```

5.1.6 delattr

Syntax

```
delattr(node parent, string sName);
delattrexpr(node parent, string sExpression);
```

Description

The first form deletes the attribute with the given name from the given `parent` node. If the attribute is not found, no error occurs.

The second form deletes all attributes found using the XPath expression passed, relative to the `parent` node.

Example

```
delattr(xf, "scope");
```

Deletes the attribute named "scope" from the node given in `xf`.

```
delattrexpr(xf, "*/@scope");
```

Deletes the attribute named "scope" from all immediate children of the node given in `xf`, irrespective of the name of those child elements.

5.1.7 addnode

Syntax

```
node = addnode(node parent, string sTag);
```

5.1.8 delnode

Syntax

```
delnode(node nNode);
delnodes(nodelist nNodes);
delnodeexpr(node nParent, string sExpression);
```

5.1.9 settext

Syntax

```
settext(node nParent, string sText);
```

Description

Adds or replaces the element text for the given node.

5.1.10deltext

Syntax

```
deltext(node nParent);
```

Description

If the node `nParent` has a text element, that text element is removed. If it does not have a text element, no error occurs.

5.1.11getnode

Syntax

```
node getnode(node parent, string expr);
```

Description

The expression `expr` is applied to the node `parent` to select one node. If a list of nodes results from the selection, only the first node is returned. If you want the whole list, use `getnodes()` instead. The `expr` must be a valid XPath expression.

Example

```
node xf = xmlopen("my.xml");
node n = getnode(xf, "object[@scope='public']");
```

After the call, `n` will contain the first node under the root element having the tag "object" and also having an attribute "scope" with the value "public".

See also

`getnodes()`

5.1.12getnodes

Syntax

```
nodelist getnodes(node parent, string expr);
```

Example

```
open xml "test.xml" as xf;
nodelist attribs = getnodes(xf, "attrib");
```

Since the file label "xf" refers to the root element, the function call will return a list of all "attrib" elements under the root element in the given xml file.

See also

`getnode()`

5.1.13transform

Syntax

```
transform(node xmlroot, node xslroot);
```

Description

Applies the XSLT template pointed to by the xslroot parameter to the XML document pointed to by the xmlroot parameter.

5.2 File related functions

5.2.1 filedelete

Syntax

```
filedelete(string sFileName);
```

Description

If the file given by `sFileName` exists, it is deleted from disk. If it does not exist, no error occurs.

5.2.2 filemove

Syntax

```
filemove(string sOldName, string sNewName);
```

Description

Will move or rename either a file or a directory (including its children) either in the same directory or across directories. It will fail to move directories (but not files) if source and destination are on different volumes.

5.2.3 filecopy

Syntax

```
filecopy(string sOriginal, string sCopy);
```

Description

Copies files (not directories) from one location and name to another. If the destination file already exists, the call will fail.

Example

```
filedelete(sCopy);
filecopy(sOriginal, sCopy);
```

Since filedelete() does not cause runtime errors if the file does not exist, this example shows the safest way of copying files over possibly pre-existing files.

5.2.4 dir

Syntax

```
set files = dir(string sMask)
```

Description

Finds all filenames matching the given mask and returns them as a set of strings. If no matching files are found, an empty set is returned.

Example

```
set fl = dir("c:\projects\cactus\x2c\inputs\*.xml");
```

The names of all xml files in the given directory will be stored in the set variable fl.

5.2.5 getcurdir

Syntax

```
string sCurDir = getcurdir();
```

Description

Retrieves the current working directory as a string. The default working directory when the interpreter is started is the directory it was started from at the command line, not necessarily the directory the executable file is located.

See also

```
setcurdir()
```

5.2.6 setcurdir

Syntax

```
setcurdir(string sCurDir);
```

Description

Sets the current working directory. This is the directory that will be used for output files created with the `open` statement, if no path, or a relative path, is supplied during the open.

See also

```
getcurdir()
```

5.3 String functions

5.3.1 fileext

Syntax

```
string fileext(string s);
```

Example

```
string s = fileext("c:\program files\cactus\x2c.exe";
```

s will contain the string: `"exe"`.

See also

filename, filepath

5.3.2 filename

Syntax

```
string filename(string s);
```

Example

```
string s = filename("c:\program files\cactus\x2c.exe";
```

s will contain the string: `"x2c"`;

See also

fileext, filepath

5.3.3 filepath

Syntax

```
string filepath(string s);
```

Description

Determines the part of s that contains drive and folder path, up to and including any trailing backslash or colon.

Return value

The drive:path part of string s. An empty string is returned if no path separators were found.

Example

```
string s = "c:\program files\cactus\x2c.exe";
string p = filepath(s);
```

p will contain the string: `"c:\program files\cactus\"`.

See also

`filename(), fileext()`

5.3.4 lcase

Syntax

```
string lcase(string s);
```

Description

Returns the string in the parameter with all alpha characters converted to lower case.

See also

`ucase()`

5.3.5 ucase

Syntax

```
string ucase(string s);
```

Description

Returns the string in the parameter with all alpha characters converted to upper case.

See also

`lcase()`

5.3.6 replace

Syntax

```
string replace(string sIn, string sFind, string sNew);
```

Description

Replaces all occurrences of string sFind in string sIn with string sNew.

Example

```
string sFileName = "myfile.h";
string sDefine = ucase(replace(sFileName, ".", "_"));
```

After processing the above, sDefine will contain the string "MYFILE_H".

5.3.7 dropchars

Syntax

```
string dropchars(string sIn, string sChars);
```

Description

Removes all occurrences of each character in sChars from the string sIn and closes the gaps.

Example

```
string sOut = dropchars('{"ABC","DEF"}', '{}"');
```

results in sOut containing:

```
ABC,DEF
```

```
set Tables;
Tables += "Doctors";
Tables += "Nurses";
Tables += "Patients";
string sSetRep = Tables;
string sTabRep = dropchars(Tables, '{}"');
```

Results in the strings holding the following contents:

```
sSetRep:  {"Doctors","Nurses","Patients"}
sTabRep:  Doctors,Nurses,Patients
```

5.4 map and set functions

5.4.1 unmap

Syntax

```
string unmap(string mapname, string key);
string unmap(string mapname, double key);
```

Description

The unmap function is used to access the values in maps. You call the unmap function passing it the name of the map and the key value to look for. According to the contents of the map and the value of the `default` clause, the unmap function returns a new string value, the key value from the second parameter or raises a runtime error.

See also

Data types: `map`.

5.4.2 intersect

Syntax

```
set = intersect(set s1, set s2);
```

Description

The function returns a set containing all strings that are present in *both* the input sets.

Example

```
set s1 = {"a", "b", "c"};
set s2 = {"b", "c", "d"};
set s3 = intersect(s1, s2);
set s4 = intersect(s2, s1);
```

`s3` and `s4` will both contain the same set, namely: `{"b", "c"}`.

5.5 System functions

5.5.1 datetime

Syntax

```
string datetime(string sFormat);
```

Description

Returns current date and time in a formatted string suitable for insertion into generated files as a constant, comment or similar.

The *sFormat* argument consists of one or more codes preceded by a percent sign (**%**). Characters that do not begin with **%** are copied unchanged to the returned string. The current locale affects the output formatting.The formatting codes are listed below. If an empty string is passed as argument, the following format string will be automatically applied: `"%A %d %b %Y %H:%M:%S".`

%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%d	Day of month as decimal number (01 – 31)

%H	Hour in 24-hour format (00 – 23)
%I	Hour in 12-hour format (01 – 12)
%j	Day of year as decimal number (001 – 366)
%m	Month as decimal number (01 – 12)
%M	Minute as decimal number (00 – 59)
%p	Current locale's A.M./P.M. indicator for 12-hour clock
%S	Second as decimal number (00 – 59)
%U	Week of year as decimal number, with Sunday as first day of week (00 – 53)
%w	Weekday as decimal number (0 – 6; Sunday is 0)
%W	Week of year as decimal number, with Monday as first day of week (00 – 53)
%x	Date representation for current locale
%X	Time representation for current locale
%y	Year without century, as decimal number (00 – 99)
%Y	Year with century, as decimal number
%z, **%Z**	Time-zone name or abbreviation; no characters if time zone is unknown
%%	Percent sign

A # flag may prefix any formatting code. In that case, the meaning of the format code is changed as follows.

Format Code	**Meaning**
%#a, **%#A**, **%#b**, **%#B**, **%#p**, **%#X**, **%#z**, **%#Z**, **%#%**	# flag is ignored.
%#c	Long date and time representation, appropriate for current locale. For example: "Tuesday, March 14, 1995, 12:41:29".
%#x	Long date representation, appropriate to current locale. For example: "Tuesday, March 14, 1995".
%#d, **%#H**, **%#I**, **%#j**, **%#m**, **%#M**, **%#S**, **%#U**, **%#w**, **%#W**, **%#y**, **%#Y**	Remove leading zeros (if any).

Example

```
.// Autogenerated on <$datetime("%a %d %B %Y %H:%M:%S")>
```

would generate an output line similar to:

```
// Autogenerated on Wed 27 march 2002 13:45:03
```

5.6 Debug and help functions

5.6.1 message

Syntax

```
message(string msg);
```

Description

The string (or string expression) is output to the console during processing. A carriage return is appended by the runtime.

6 Examples

Several real world examples of templates are presented here.

6.1 C++ accessor template

This example takes an xml description of a data layer object and writes the C++ code for an OLEDB consumer accessor object, very similar to what the Visual C++ OLEDB consumer wizard produces. To save space, only the header file is produced in this example. To save space and minimize line-wraps, I'm using a fairly small font, but you can fix that with big glasses.

XML object definition

```
<?xml version='1.0'?>
<package name='a_s'>
  <object name='Ambulancier'>
    <primkey>
      <column name='Key'/>
    </primkey>
    <attribute name='Changedate' table='AMBULANCIERS' column='A_CHANGEDATE'
        type='date' dbtype='DATE' is-required='no' magic='changedate' scope='private'/>
    <attribute name='Changedby' table='AMBULANCIERS' column='A_CHANGEDBY'
        type='string' dbtype='VARCHAR2' is-required='no' magic='changedby'
        scope='private' dbsize='10'/>
    <attribute name='Code' table='AMBULANCIERS' column='A_CODE' type='string'
        dbtype='VARCHAR2' is-required='yes' scope='public' dbsize='10'/>
    <attribute name='Dateinvalid' table='AMBULANCIERS' column='A_DATEINVALID'
        type='date' dbtype='DATE' is-required='no' scope='public'/>
    <attribute name='Key' table='AMBULANCIERS' column='A_KEY' type='double'
        dbtype='NUMBER' is-required='yes' scope='public' dbsize='10'/>
    <attribute name='Name' table='AMBULANCIERS' column='A_NAME' type='string'
        dbtype='VARCHAR2' is-required='no' scope='public' dbsize='30'/>
    <attribute name='Soundex' table='AMBULANCIERS' column='A_SOUNDEX' type='string'
        dbtype='VARCHAR2' is-required='no' scope='public' dbsize='30'/>
  </object>
</package>
```

x2c script

```
// accessor.tpl
//
// this template file uses an xml description of a data layer object
// with its corresponding database table/column definitions and produces
// a c++ header file containing the OLEDB accessor code for it. It is derived
// from code as generated by the VC++ wizard for OLEDB consumers. I've not
// tested the produced code, since it's only intended to illustrate what kind
// of stuff you can generate.
//
//


map cpptype = {
        "string"        :       "TCHAR",
        "date"          :       "DBTIMESTAMP",
        "double"        :       "DB_NUMERIC",
        default         :       "error"
};

map comtype = {
        "string"        :       "BSTR",
        "date"          :       "DATE",
        "number"        :       "DOUBLE",
        "double"        :       "DOUBLE",
        "long"          :       "LONG",
        default         :       error
};
```

```
string sOutputDir = "out\";
set files = dir('*.xml');

foreach string sFN in files
{
        string sBaseName = filename(sFN);
        string sOutputHFile = sOutputDir & sBaseName & ".h";
        string sOutputCPPFile = sOutputDir & sBaseName & ".cpp";
        node root = xmlopen(sFN);
        writeHFile(root, sOutputHFile);
        writeCPPFile(root, sOutputCPPFile);
};

// ========================================================================
// write a .H file
// ========================================================================

function writeHFile(node root, string sOutputFileName)
{
        open create sOutputFileName;
        string sDefine = ucase(filename(sOutputFileName)) & "_H";

        .// <$sOutputFileName>
        .// Autogenerated H file
        .
        .#ifndef <$sDefine>
        .#define <$sDefine>

        string sClassName = "C" & evalattr(getnode(root, "object"), "name");
        string sAccessorName = sClassName & "Accessor";
        .class <$sAccessorName>
        .{
        .public:

        nodelist attribs = getnodes(root, "object/attribute[@scope='public']");
        foreach node n in attribs // a
        {
                string sAttrName = evalattr(n, "name");
                string sCppType = unmap("cpptype", evalattr(n, "type"));
                double dblDbSize = evalattr(n, "dbsize");

                if (dblDbSize > 0)
                {
                .       <$sCppType>    m_<$sAttrName>[<$dblDbSize+1>];
                }
                else
                {
                .       <$sCppType> m_<$sAttrName>;
                }
        };


        .
        .BEGIN_COLUMN_MAP(<$sAccessorName>)

        double dblIndex;
        foreach node n in attribs // b
        {
                dblIndex++;
        .       COLUMN_ENTRY(<$dblIndex>, m_<$evalattr(n, "name")>)
        };
        .END_COLUMN_MAP()
        .
        .DEFINE_COMMAND(<$sAccessorName>, _T(" \
        .       SELECT \

        set sTables;

        foreach node n in attribs // c
        {
        .       <$evalattr(n, "column")>, \
                sTables = setadd(sTables, evalattr(n, "table"));
        };
        [[      FROM]]

        double notfirst;
```

```
        foreach string sTable in sTables
        {
                if (notfirst != 0) [[,]]
                [[ SYSADM.<$sTable>]]
                notfirst++;
        };
        [["))]]
        .


        // here follows a chunk of almost verbatim code output

[[
        // You may wish to call this function if you are inserting a record
        // and wish to initialize all the fields, if your are not going to
        // explicitly set all of them.
        void ClearRecord()
        {
                memset(this, 0, sizeof(*this));
        }
};

class <$sClassName> : public CCommand<CAccessor<<$sAccessorName>> >
{
public:
        HRESULT Open()
        {
                HRESULT        hr;

                hr = OpenDataSource();
                if (FAILED(hr))
                        return hr;

                return OpenRowset();
        }

        HRESULT OpenDataSource()
        {
                HRESULT        hr;
                CDataSource    db;
                CDBPropSet     dbinit(DBPROPSET_DBINIT);

                dbinit.AddProperty(DBPROP_AUTH_PASSWORD, OLESTR("sysadm"));
                dbinit.AddProperty(DBPROP_AUTH_USERID, OLESTR("sysadm"));
                dbinit.AddProperty(DBPROP_INIT_DATASOURCE, OLESTR("hosp2"));
                dbinit.AddProperty(DBPROP_INIT_LCID, (long)2067);
                dbinit.AddProperty(DBPROP_INIT_PROMPT, (short)4);
                hr = db.Open(_T("MSDAORA.1"), &dbinit);
                if (FAILED(hr))
                        return hr;

                return m_session.Open(db);
        }

        HRESULT OpenRowset()
        {
                return CCommand<CAccessor<<$sAccessorName>> >::Open(m_session);
        }
        CSession       m_session;
};
#endif

]]

}

// ========================================================================
// write a .CPP file
// ========================================================================

function writeCPPFile(node root, string sOutputFileName)
{
        open create sOutputFileName;
        .// <$sOutputFileName>
        .// Autogenerated CPP file
        .
        .#include "stdafx.h"
```

```
        .#include "<$filename(sOutputFileName) & '.h'>"
        .

};

// ============================================================================
```

C++ output header file

```
// out\Ambulancier.h
// Autogenerated H file

#ifndef AMBULANCIER_H
#define AMBULANCIER_H
class CAmbulancierAccessor
{
public:
        TCHAR   m_Code[11];
        DBTIMESTAMP m_Dateinvalid;
        DB_NUMERIC     m_Key[11];
        TCHAR   m_Name[31];
        TCHAR   m_Soundex[31];

BEGIN_COLUMN_MAP(CAmbulancierAccessor)
        COLUMN_ENTRY(1, m_Code)
        COLUMN_ENTRY(2, m_Dateinvalid)
        COLUMN_ENTRY(3, m_Key)
        COLUMN_ENTRY(4, m_Name)
        COLUMN_ENTRY(5, m_Soundex)
END_COLUMN_MAP()

DEFINE_COMMAND(CAmbulancierAccessor, _T(" \
        SELECT \
        A_CODE, \
        A_DATEINVALID, \
        A_KEY, \
        A_NAME, \
        A_SOUNDEX, \
        FROM SYSADM.AMBULANCIERS"))

        // You may wish to call this function if you are inserting a record
        // and wish to initialize all the fields, if your are not going to
        // explicitly set all of them.
        void ClearRecord()
        {
                memset(this, 0, sizeof(*this));
        }
};

class CAmbulancier : public CCommand<CAccessor<CAmbulancierAccessor> >
{
public:
        HRESULT Open()
        {
                HRESULT        hr;

                hr = OpenDataSource();
                if (FAILED(hr))
                        return hr;

                return OpenRowset();
        }

        HRESULT OpenDataSource()
        {
                HRESULT        hr;
                CDataSource    db;
                CDBPropSet     dbinit(DBPROPSET_DBINIT);

                dbinit.AddProperty(DBPROP_AUTH_PASSWORD, OLESTR("sysadm"));
                dbinit.AddProperty(DBPROP_AUTH_USERID, OLESTR("sysadm"));
                dbinit.AddProperty(DBPROP_INIT_DATASOURCE, OLESTR("hosp2"));
                dbinit.AddProperty(DBPROP_INIT_LCID, (long)2067);
                dbinit.AddProperty(DBPROP_INIT_PROMPT, (short)4);
                hr = db.Open(_T("MSDAORA.1"), &dbinit);
                if (FAILED(hr))
```

```
                        return hr;

                return m_session.Open(db);
        }

        HRESULT OpenRowset()
        {
                return CCommand<CAccessor<CAmbulancierAccessor> >::Open(m_session);
        }
        CSession        m_session;
};
#endif
```

6.2 Object – link – db example

A fairly natural way of defining a data access layer would be to maintain three different kinds of xml files:

1. Object definitions as seen by the higher business layer. These definitions can be in a file per object or collected into one or a few xml files holding many objects.
2. Database table definitions. These should all be collected into one table to avoid excessive file manipulations during searches.
3. Links between the object definitions and the database table definitions. Since these links are also used during lookups, they should be collected into a single xml file.

In the sample, there's only one object, thus one object definition file and its corresponding link and database information. The files are called obj.xml, links.xml and db.xml and can be found in the directory ".\samples\linked". To keep this sample short, it only produces a very brief output file that certainly isn't a functional data access object in any language but that allows us to demonstrate how to look up information via a link file.

For this example, only the script code is given below. All the files can be found in the folder `samples\linked` under the x2c installation folder.

linked.x2c script file

```
// linked.x2c
//
// illustrates using object definition, link file and db description
//
// for each object in an object definition file, each attribute is used for output,
// and a link is accessed in the link file. That link then leads to something in
// the db definition allowing the full database access to be written.
//
// The problem we solve here is looking up data from the primary xml file using
// one or more secondary xml files.
//
// To do this, the links and the db definitions need to be collected
// into a link file and a db definitions file. If the links and/or
// the definitions are spread out over a number of xml files, the problem is
// entirely much worse and not treated here. The objects may or
// may not be in a file per object or a single file holding all objects. In this
// example, there's only one object in the obj.xml file, so it's a moot point.

map cpptype = {
        "string"        :       "TCHAR",
        "date"          :       "DBTIMESTAMP",
        "double"        :       "DB_NUMERIC",
        default         :       "error"
};
```

```
map comtype = {
        "string"        :       "BSTR",
        "date"          :       "DATE",
        "number"        :       "DOUBLE",
        "double"        :       "DOUBLE",
        "long"          :       "LONG",
        default         :       error
};



// open all input files first

node fObj = xmlopen(getcurdir() & '\obj.xml');
node fLinks = xmlopen(getcurdir() & '\link.xml');
node fDb = xmlopen(getcurdir() & '\db.xml');

// get all the object definitions into a list of nodes

nodelist nsObj = getnodes(fObj, 'object');


// now do the objects one by one

foreach node nObj in nsObj
{
        writeHFile(nObj);
};



// ============================================================================
// nObj - current object in object file to write
//
function writeHFile(node nObj)
{
        string sOutputFileName = makeOutputFileName(nObj, ".h");
        open create sOutputFileName;

        string sDefine = ucase(replace(sOutputFileName, '.', '_'));

        // write boilerplate part of output file
        .//
        .// <$sOutputFileName>
        .// Autogenerated by x2c on <$datetime('')>
        .
        .#ifndef <$sDefine>
        .#define <$sDefine>
        .
        string sClassName = "C" & evalattr(nObj, "name");
        string sAccessorName = sClassName & "Accessor";
        .class <$sAccessorName>
        .{
        .public:

        // select all the object attributes designated with scope public
        nodelist nlPubs = getnodes(nObj, "attribute[@scope='public']");

        // for each attribute, add a data member in the public section
        foreach node nPub in nlPubs
        {
                // get the name of the attribute
                string sPubName = evalattr(nPub, "name");

                // convert its attribute type to something C++ understands
                string sCppType = unmap("cpptype", evalattr(nPub, "type"));

                // get its size from the database description
                double dblSize = getDbSize(evalattr(nObj, "name"), sPubName);

                // depending on if it has a size or not,
                // the members are declared differently
                if (dblSize > 0)
                {
                .       <$sCppType>    m_<$sPubName>[<$dblSize + 1>];
                }
```

```
                else
                {
                .       <$sCppType>    m_<$sPubName>;
                }
        }

        // add the declaration of the column map
        .
        .BEGIN_COLUMN_MAP(<$sAccessorName>)

        double dblIndex;
        foreach node nPub in nlPubs
        {
                dblIndex++;
        .       COLUMN_ENTRY(<$dblIndex>, m_<$evalattr(nPub, 'name')>)
        };
        .END_COLUMN_MAP()
        .
        .DEFINE_COMMAND(<$sAccessorName>, _T("SELECT \

        set setTables;
        string sObjName = evalattr(nObj, "name");
        foreach node nPub in nlPubs
        {
                string sAttrName = evalattr(nPub, "name");
                string sColName = getDbColName(sObjName, sAttrName);
        .       <$sColName>, \
                setTables += getDbTableName(sObjName, sAttrName);
        }
        // get the string form of the set of tables and remove
        // the braces and quotes from it
        string sTables = dropchars(setTables, '{}"');
        [[      FROM <$sTables>"))]]
        .

        .// here we excluded the rest of the accessor code, since it's the same
        .// as in the first example

}

// ============================================================================
// make a suitable name for the output file
function makeOutputFileName(node nObj, string sExt) : string
{
        string sObjName = evalattr(nObj, 'name');
        return sObjName & sExt;
}

// ============================================================================
// retrieve database column size for object/attribute
function getDbSize(string sObjName, string sAttrName) : double
{
        string sTableName = getDbTableName(sObjName, sAttrName);
        string sColName = getDbColName(sObjName, sAttrName);

        string sDbSelect = "table[@name='" &
                sTableName & "']/column[@name='" & sColName & "']";
        node nDb = getnode(fDb, sDbSelect);
        return evalattr(nDb, 'dbsize');
}

// ============================================================================
// retrieve database column name for object/attribute
function getDbColName(string sObjName, string sAttrName) : string
{
        string sSelect = "object[@name='" &
                sObjName & "']/map[@attrname='" & sAttrName & "']";
        node n = getnode(fLinks, sSelect);
        return evalattr(n, "column");
}

// ============================================================================
// retrieve database table name for object/attribute
function getDbTableName(string sObjName, string sAttrName) : string
{
        string sSelect = "object[@name='" &
                sObjName & "']/map[@attrname='" & sAttrName & "']";
```

```
        node n = getnode(fLinks, sSelect);
        return evalattr(n, "table");
}
```

Index