Extensions to Universal Pointer Capabilities
June, 2004

In SLX, a universal pointer (pointer(*)) can be assigned a pointer to any class of object. Until now, universal pointers could be used only for *storing* pointers to objects, but not for *accessing* attributes of the objects to which they pointed. This restriction was imposed because SLX could not handle attributes with the same name, but contained in multiple classes. Consider the following example:

```
class widget1
    {
    int       i;
    double    x;
    int       x1,
    };

class widget2
    {
    double    x;
    int       i;
    int       x2;
    };

class widget3
    {
    booleanbusy;
    };

pointer(*)    u;
...
u -> i = 999;
u -> x = 2.5;
u -> busy = TRUE;
```

The classes widget1 and widget2 two each contain an integer attribute named “i” and a double attribute named “x”. Class widget3 contains neither an “i” nor an “x”. Thus, compiling code for run-time evaluation of “u -> x” is extremely difficult. For this reason, until now SLX disallowed such usage.

SLX has gotten smarter. Universal pointers can now be used to access objects’ attributes.

If a universal pointer is used to reference an attribute contained in one and only one class, processing the reference is easy. In the above example, “u -> busy” is valid if and only if “u” points to an object of class widget3.

Processing of “u -> x” and “u -> i” is considerably more difficult, for two reasons. First, “u” must point to an object of class widget1 or widget2, since widget3 contains neither attribute. Second, the offsets of “i” and “x” within their containing objects depend on whether they’re contained in a widget1 or a widget2.

The SLX compiler now recognizes attributes that are common to multiple classes, and it generates tables of attribute offsets that are used during execution to resolve references made by means of universal pointers to such attributes. The code SLX generates to resolve such references is a fixed sequence of machine instructions containing no looping. The tables are directly indexed; lookup *searches* are not required. The additional complexity beyond class-

specific pointer evaluation is minimal. (on the order of a nanosecond or so). Thus, evaluating a million such references takes a millisecond or so.

To be considered attributes common to multiple classes, variables must meet the following requirements:

1. They must have the same name.
2. They must be of the same type, e.g. double.
3. They must have identical prefixes. For example, one cannot be a control variable and the other not. One cannot be public and the other private, etc.
4. Arrays must have the same *number* of dimensions. For example, a 2-dimensional array and a 3-dimensional array are incompatible.
5. If enumerated types are used as array dimensions, the same types must be used for the same dimensions.
6. Except for requirement 5, dimension *values* can differ. For example “int ix[2][2]” and “int ix[3][3]” are compatible.
7. Common attributes cannot be static. (Static attributes are stored in global locations shared by all members of their class. Because static attributes are not stored in each instance of a class, their “offset from beginning of object” is undefined.)

If you elect to use universal pointers to access object attributes, please exercise caution, because errors that formerly would have been caught at compile time now must be caught at run time. If there’s a possibility that “u -> x” could be valid, detecting instances where “u” points to the wrong kind of object or is NULL can only be done at run time.

You may find it helpful to use a common prefix in the names of common attributes, e.g.,

CA_xloc,
CA_yloc,
CA_speed

Take a look at the Wolverine-provided example named comnattr.slx. Study it carefully. It contains a number of lines that are commented out. Un-commenting these lines will result in compile-time or run-time errors. You will find looking at a few of these errors instructive.

A listing of comnattr.slx follows on the next page.

```
//*****************************************************************************
//      Common Attributes
//*****************************************************************************

public precursor module MoreDifficult
        {
        typedef enum { RED, GREEN, BLUE }     color;
        typedef enum { ON, OFF }              state;

        class widget
                {
                control int  i;
                int          j, unique1;
                double       x, y;
                string(10)   s1;
                string(20)   s2;

                control int  array1[2][3][4];
                int          array2[2][3][4];
                int          array3[2][color][4];

                color        c1, c2;
                state        sw1, sw2;
                };

        procedure test_uptr(pointer(*) u, int answer)
                {
                yield;
                print(u -> i, answer)       "In test\_uptr: _ == _\n";
                return;
                }
        }

public module CommonAttributes
        {
        class widget2
                {
                double       x, j, unique2;             // common x
                control int  i, y;                      // common i
                string(30)   s2;
                string(10)   s1;                        // common s1

                control int  array1[2][3][4];
                int                  array2[2][4][3];           // different dimensions from class widget
                int                  array3[2][state][4];

                color        c1, sw2;                   // common c1
                state        sw1, c2;                   // common sw1
                };

        class widget3
                {
                boolean      not_much_here;
                };
```

```
procedure main()
        {
        pointer(widget)    w;
        pointer(widget2)  w2;
        pointer(*)                  u;

        w = new widget;

        w -> unique1 = 111;
        w -> i = 101;
        w -> j = 102;
        w -> x = 101.0;
        w -> y = 102;
        w -> s1 = "w1 s1";
        w -> s2 = "w1 s2";

        w -> array1[2][2][2] = 111;
        w -> array2[2][2][2] = 111;
        w -> array3[2][BLUE][2] = 111;

        w2 = new widget2;

        w2 -> unique2 = 222.0;
        w2 -> i = 201;
        w2 -> j = 202;
        w2 -> x = 201.0;
        w2 -> y = 202;
        w2 -> s1 = "w2 s1";
        w2 -> s2 = "w2 s2";

        w2 -> array1[2][2][2] = 222;
        w2 -> array2[2][2][2] = 222;
        w2 -> array3[2][OFF][2] = 111;

        u = w;

        print(w -> i, u -> i)              "_ == _\n";
        print(w -> x, u -> x)       “_._ == _._\n”;
        wait until (w -> i != 0);
        wait until (u -> i != 0);
        wait until (w -> array1[2][2][2] != 0);
        wait until (u -> array1[2][2][2] != 0);

        test_uptr(u, 101);

        print(w -> s1, u -> s1)    "_ == _\n";

//      print(w -> j, u -> j)         "_ == _\n";          // compile-time error: incompatible j's

        print(u -> array1[2][2][2])              "array 111 == _\n";
        print(u -> array2[2][2][2])              "array 111 == _\n";

//      print(u -> array3[2][BLUE][2])  "array 111 == _\n";        // incompatible array3

//      u = w2;                                          // causes run-time error in next statement
```

```
            print(u -> unique1)       "111 == _\n";       // "unique1" is only in class widget

            u = w2;
//          u = NULL;                 // causes run-time error below
//          u = w;                          // ditto (wrong class for unique2)

            print(u -> unique2)       "222.0 == _._\n";  // "unique2" is only in class widget2

            print(u -> x)             "201.0 == _._\n";

            print(u -> array1[2][2][2])   "array 222 == _\n";
            print(u -> array2[2][2][2])   "array 222 == _\n";

            test_uptr(u, 201);  // subtlety: widget2 unknown when test_uptr compiled in precursor

            u = new widget3;
//          u -> i = 999;             // run-time error: no "i"
            test_uptr(u, 0);          // run-time error: no "i"

            exit(0);
            }
        }
```