The Luban Programming Language

Peter X. Huang

12/2005

Contents

1 What Is Luban

Windows were shakin' all night in my dreams
Everything was exactly the way it seems
- Highlands, Bob Dylan

Luban is a component oriented programming language. Luban is named to honor a legendary ancient Chinese civil engineer two thousand year ago whose constructions are still in use today.

Luban, in short speaking, is a scripting language with a robust component and composition model. Luban is easy to use. To start coding Luban requires almost zero understanding of any high level programming concepts. If you know what you want to do plus a minimum set of Luban's basic features, you can code your work with ease. Luban's syntax is simple and clean. And Luban code requires no explicit compiling and linking steps to run. If you like the usability of scripting language, you will like Luban as well.

Intuitiveness and usability are the design principles of Luban. And that is why Luban has the unique component composition feature. Component composition is a technology that has been used in engineering. Yet Luban is the first programming language that directly supports component composition in software construction. Luban component composition is analogous to spreadsheet construction. User defines a new component by laying down sub-components and specifying dependencies among the sub-components. Composition fits a range of problems that could be awkward to code in traditional procedural way. More importantly, composition is simple and intuitive. Any one who can use a spreadsheet can immediately understand Luban component composition.

Luban also fully supports component oriented programming paradigm in which users can easily share and reuse component. Luban's component design is similar to Java Bean, yet much more robust. You may ask that why not use object oriented paradigm. The answer is object oriented design is probably too complicated for general scripting language users. Scripting language needs its own component model to go further than coding ad hoc short script, and its component model should be simpler than class.

Luban's component features include component interface definition and inheritance, component name space, built-in multi-threading, dynamic component loading and dynamic type checking. You can start using Luban only for scripting and later may find these component features handy when you try to collect your Luban scripts into one or more coherent systems.

In conclusion, Luban is an integration layer programming language that has both the usability of scripting language and the scalability of structured language. The goal of Luban is to present a simple component oriented programming tool to programmers and a potentially much wider audience that may never have been able to program before. The design is to empower people who have the field knowledge to directly code and share their knowledge in the form of software components, no matter if they are professional programmer or not.

2 Start Scripting Luban

In the beginner's mind there are many possibilities,
in the expert's mind there are few.
- Shunryu Suzuki, Zen Master

This chapter introduces the basic procedural code constructions in Luban through examples. This chapter is all you need to read for basic Luban scripting. The syntax of Luban procedural code is very much like simplified C++ or Java. Basically a valid Luban script is a series of one or more Luban statements, separated by either semicolon or bracket. Change of line doesn't mean much for Luban. So in theory you can put all your code in one big line and claim you finish a system in a single line of code.

2.1 Hello World

Nothing better illustrates a programming language than code examples. And all self-respecting languages start with a "Hello, world" example. Luban is no exception.

```
std::println(obj="Hello, world!");
```

Type above line into a file named "helloworld.lbn". Then you can run your first Luban program by type the following in your command prompt of your operating system:

```
Mycomputer > luban helloworld.lbn
Hello, world!
```

This simple "Hello, world" Luban example code does one thing that is to call one Luban component structure of type *std::println* with its input property "obj" set to string "Hello, world!". This component call trigger the evaluation of std::println that prints out the "Hello, world!" message on console screen..

2.2 Procedural Statements

The basic procedural constructions of Luban consist of expressions and statements. If you know C++ or Java, you can skip most of this section except the *foreach* statement and multi-threading part.

2.2.1 Expression and Assignment

The most basic Luban statement can be an expression or an assignment statement. Listed below are different kinds of Luban expressions, in the order of precedence.

Constants:

```
1; 3.14; true; false; "Hello"; 'a';
```

Variables are also expressions. Luban's variable name rule is the same as C/C++/Java

```
varx; _varx_; v123;
```

Data object and component construction:

```
[ 1,2,3, x, y, z, a+b ];  // vector construction
{ "one": 1, "two" : 2 }; // map construction
{ "one", "two" };   // set construction
```

double("3.1415"); // double type construction
std::file("mydatafile", 'r'); // open file
mynamespace::mycomponentX(); //Luban component construction
mynamespace::mycomponetX(prp1 = 100); // component call

Container element access, data object member function call, component property access and component object call.

mymap["one"]; // map indexing
x[1] = y[2]; // vector or map indexing
mymap.remove("one"); // data object member function call
linevector = allines.split('\n'); // data object member function call
mycalculator(input1=1.1, input2=2.2); // component object call

Single operand operators, ++ -- and negation:

++i; --i; i++; i--; // i will be changed
y = -x; // x is not changed, y get the –x result;

Common arithmetic operations:

*1+2; 10.0*3.0; 50%3; 50/3; "Hello, "+"world";*

Type checking operations:

objx isa string;
typeof(objx) == string;

Equality and order checking:

x == y; x != y; x > y; x < y; x >= y; x <= y;

Logical expressions:

true or false; x and y; not true; not a and not b;
true || false; x && y; ! true; ! a && !b; // for people who know C/C++/Java

Conditional expression:

x > y ? x: y;

Assignment statements:

x = 1; x += 1; x[index] = 2; x.property = 3; x=2; x/=3;*

The details of some of the expressions like property reference and component call can be found in later chapters.

2.2.2 Flow Control

Luban has a set of common flow control statements. The below is a brief list and examples of usage.

IF statement:

if (a>b) result=a;

IF-ELSE statement:

if (a>b) result = a; else *result = b;*

WHILE statement:

while (x < 100) ++ x;

FOR statement:

for(i=0; i<100; ++i) std::console().writeline(vec[i]);
for(;;); // forever loop

FOREACH statement:

foreach(key, value in {"one":1, "two":2})

```
        std::console().writeline(key, ' ', value);
    foreach( element in [ 1,2,3 ] )
        std::console().writeline(element);
```

BREAK statement, used to jump out of a loop:

```
    for( i=0; ; i++) if ( i>100 )  break;
```

CONTINUE statement, used to jump to the beginning of a loop:

```
    for( i=0; ; i++) if ( i<=100 )  continue; else break;
```

DONOTHING statement:

```
    ;
```

FINISH statement, terminate the code execution in current component.

```
    finish;
```

WAIT statement, wait for all dispatched threads to finish.

```
    wait;
```

WAITFOR statement, wait for thread(s) associated with specific variable(s) to finish.

```
    waitfor x,y,z;
```

CANCEL statement, cancel and join all dispatched threads

```
    cancel;
```

Details for these thread related statements can be found later in this chapter.

2.2.3 Compound Statement:

Using bracket *{}* to collect one or more statement together makes a compound statement. You can use compound statement anywhere you use a regular statement. For example:

```
    while ( x <100 or y < 200 )  { x++; y++; }
```

2.2.4 Component Property Batch Setting

You can set multiple properties of a component in one statement, as below:

```
    astruct.(prp1=1, prp2=2, prp3=3);
```

2.2.5 Comments

To put comments into Luban code is the same as C++. Anything from *//* to the end of line are comments. Anything between a */** to a ***/** are comments too. For example:

```
    ; // This line does nothing
    ; /* this line does nothing either */
```

2.3 Luban Built-In Data Types

Luban has a set of native data types to support most common operations. Below describes the native data types and their relations.

2.3.1 Double and Integer:

The key words for double (double precision floating point number) and integer types are *double* and *int* in Luban. Their relationship is similar to the double and int type in C. As below examples:

```
    10/3;           // result is integer 3
    10.0/3.0;       // result is double 3.333333…
    10/3.0;         // result is double 3.3333… mix double and int, result in double
    0 == 0.0;       // result is true, you can mix and compare double and int
```

```
2 > 0.1;          // result is true, you can mix and compare double and int

x=1.23456789;
y=x.format(".4"); // y= "1.2346" format to 4th decimal place
z = int(x);   // z  is 1, cast double into integer
xstring = "1.23456E7";
xdouble = double(xstring);  // x = 1.23456x10^7 construct double from string
```

2.3.2 Boolean

The key word for Boolean type is *bool.* Boolean can only be *true* or *false.* And unlike C, There is no implicit casting to and from any other type. And the operators for Boolean type data are *not, and, or*. Applying arithmetic operation to Boolean type will result in error. The condition expression used in IF, WHILE and FOR statements must result in Boolean too, or the Luban code execution will stop.
One more thing to mention is that if you like, you can still use the C style &&, || and ! operators in place of *and, or, not*.

2.3.3 Character

Character type key word is *char.* It contains a single character. Below code example shows some operations on character type.

```
for( onechar ='a'; onechar <= 'z'; ++onechar)
        std::print(obj=onechar);
```

Above Luban program prints out "abcdefghijklmnopqrstuvwxyz" on screen..

```
NinetySeven = int ('a');  // NinetySeven = 97,  char to ASCII
Lettera = char(97);  // Lettera='a', ASCII to char
```

2.3.4 String

String type key word is *string*. The below example Luban code shows operations for string type.

```
s = "Hello"; c = s[1]; // result is 'e'
s = "Hello"; s[0] = 'h'; // s becomes "hello"
s = "Hello"; foreach( c in s ) std::console().write( c );  // will print out Hello
s = "Hello"*2;  // result is "HelloHello"
s = -"Hello"; // result is "olleH"
s = "Hello" + "World"; // result "HelloWorld"

// below are member function examples

// SPLIT !!
s = "Hello World"; words = s.split( ); // words is a vector ["Hello", "World"]

// trim white space from front and back
s = "     Hello";
s.trimfront();  // s becomes "Hello"
s = "Hello     \n\t";
s.trimback(); // s become "Hello"
```

```
//search
s="Hello";
index=s.find("Hell"); // index = 0
index2=s.find("Hell",2); // index = null, faled to find Hell starting from index 2l
index3=s.find("Heaven"); // index3 = null  failed to find
truth=s.contains("Hell"); // truth = true
sub=s.substr(2,3); // sub="ll" from index 2 to index 3

// content manipulation
s="Hello";
s.replace("Hell", "Heaven"); // now s becomes "Heaveno"
s.lower(); // s becomes "heaveno";
s.upper(); // s = "HEAVENO"
s.remove(6); // s="HEAVEN", with 0 at index 6 removed
s.remove("A"); // s="HEVEN";
s.insert(1, "A"); // s="HEAVEN";
s.clear(); // s=""
```

2.3.5 Vector, Map and Set

Vector type key word is *vector.*

```
x = 1; y=2; z=3; v = [x,y,z,4]; // v = [ 1,2,3,4]
v[0] = 10; // v = [10,2,3,4]
total = 0;  foreach( element in v ) total += element; // total =
v.sort(); // v become [2,3,4,10]
v2 = v+  [11,12,13]; // v2 become [2,3,4,10,11,12,13]
v2.remove(0); // v2 become [3,4,10,11,12,13]
v2.removeobj(10);  // v2 become [3,4,11,12,13]
v3 = [ v2,v2]; // v3 is two dimensional vector

// push and pop
x=[1,2];
x.pushfirst(0); // x =[0,1,2]
x.pushlast(3); // x=[0,1,2,3]
y=x.popfirst(); // y=0, x=[1,2,3];
z=x.poplast(); // y=3, x=[1,2]
```

Map type key word is *map.* Keys are unique in a map. Code examples:

```
x=1; y=2; z = 3;  word2number = { "one" : x, "two" : y,  "three" : z };
word2number["four"] = 4; // insert or replace
word2nubmer.remove("one"); // remove key "one" and associated 1
word2number.insert("two",2.0); // no impact because key already exist
union = word2number + { "five":5, "six":6};
foreach(key,val in word2number ) std::console().writeline(key,' ',val); // iterate
emptymap = {:};
```

Set key word is *set.* Set is simply a collection of unique values. See examples:

```
oneset = { 1,  “hello”, 3.14 };
another = { 1,2};
union = oneset + another;  // union = { 1,2,”hello”,3.14}
minus = oneset – another ; // minus = { “hello”, 3.14}, common values removed
joint = oneset – minus; // joint = {1}, the common values of oneset and another
foreach( val in oneset ) std::console().writeline(val);  // iterate
emptyset = {};
```

Casting operations among vector, set and map:

```
oneset = { 1,2,3}; onevec = vector(oneset);
keyvec=[‘a’,’b’]; valvec = [‘A’,’B’]; onemap = map(keyvec, valvec);
```

2.4 Common Operations For All Luban Types

All the common operations like +-*/%, [] (indexing), member function call and type casting can be applied on all Luban data types, though *foreach* statement can be applied on all Luban container data types. Though the actual results of the operations may differ depending on the data objects in the operation. There could be the case the particular operator does not apply to the data object, and the operation will result in error value in that case. It is part of Luban’s polymorphism. The same operation results differently for different data types.

2.4.1 Arithmetic Alike Operation Examples

```
//  A+B operations
numtwo = 1 + 1;   // numtwo = 2
stroneone = “1” + “1”;  // stroneone = “11”
vec = [1]+[1];   //  vec = [ 1, 1]
s = { 1 } + { 1 };   // s = {1},  set merge
m = { 1:10 }+{ 2:20};  // m = { 1:10, 2:20 }, map merge
err = “1” + 1; // err = error, can not add string and integer
err2 = true + false; // err = error can not add logical values

//A – B operations
zero = 1 – 1; // zero = 0
varset = { 1,2,3} – { 3,4,5 }; //varset = {1,2} remove common elements of two sets
err = [1]-[1];  // err = error, can not subtract vectors

// -X negation
x = -100; // x = -100
vec = - [ 1,2,3]; // vec =[3,2,1] vector negation reverses element order
str  = - “abc”;  // str = “cba” string negation is order reversing
truth = -false;  // truth = true logical value negation equals NOT operation
err = - ‘a’;  // err = error, can not negate char type

// A*B  operations
```

```
x = 2*2; // x = 4
str = "ab"*2; // str = "abab" string multiplied by integer
vec = [1,2]*2; // vec = [1,2,1,2]  vector multiplied by integer


//  A/B operations
x = 100/3;  // x = 33 integer division
xfloat = 100.0/3.0; // xfloat = 33.3333…. float number

// A%B
xmod = 100 % 3 ; // xmod = 1

// ++ operation
x = 0;
y = ++x;  // x=1, y=1, x goes up by one first, then assign to y
z = y++;  // z =1; y =2;  y assign to z first then goes up by one
x = 'a';
++x;  // x= 'b'  next of 'a' is 'b'
```

-- operator is actually the reversion of ++ operator. Below are examples for operation-and-assign operators.

```
x += 10; // same as  x = x+10;
x -= 10; // same as  x = x-10;
x /= 10; // same as  x = x/10;
x *= 10; // same as  x = x*10;
x %= 10; // same as  x = x%10;
```

It is straight forward that operation-and-assign operator is faster than the equivalent two-step operation then assign operation.

2.4.2 *foreach* Statement And Container Type

foreach statement only applies to container types including vector, map, set and string types. Applying it on non-container type will result in the termination of evaluation of the current Luban component.

```
X = 3.1415;
foreach( y in x )     // ERROR! Terminate here
        std::println(obj=y);
```

Below is an example of correct use of *foreach* statement.

```
foreach( x in [10,20,30] ) std::println(obj=x);
```

Above code will print 10,20,30, one number per line.

2.4.3 Common Member Function Calls

All Luban data object has member functions. The member functions for each data object are defined by the type of that data object. Though there are a collection of common member functions available for all Luban data types, including both built-in and user defined types. The below is the list of those common member functions and their descriptions.

```
H= x.hash();  // hash key for the object, default value 0
S= x.size();  // apply to container and string type, default value error
Y = x.contains(y); // apply to string and container type
x.clear();        // apply to string and container
guts = x.serialize(); // returns the binary guts stream in a string
h = x.invoke("hash"); // call member function hash  in a reflective way
funcmap = x.allfuncs(); //  map containing all member functions and descriptions

obj = std::des(stream=guts).obj; // restore object from guts string
```

The above last line is not a member function call. It just shows how to use the string generated from *serialize()* function call. All member function calls will return error value when error happens in the call.

2.5 Null and Error

Null is a special value in Luban, its key word is *null*. Null can be used in expressions just like a normal constant. The *null* value is special in the following two ways. First the *null* value is the default setting for all variables and properties. Second the *null* value is of no type.

Error type is a special type in Luban, with key word *error.* Value of *error* type can be generated by Luban internal operations, or explicitly constructed by Luban code. All member function calls return error value if things go wrong inside the function. For example:

```
errval = 1/0;
if ( not errval isa error )
    std::console().writeline("Your Luban engine is on fire");
expliciterr = error("Hey, I am a born error");
```

2.6 Parallel Computing: Threading and Synchronization

Multi-threading is a built-in part of Luban, as the below sample code shows.

```
myjobs::dothis(arg=100)  & // dispatch the expression in a different thread
myjobs::dothat(arg =200) & // dispatch thread
{ c.doOne();  d.doTwo(); } & // dispatch a block
wait; // wait for all three threads to finish
```

You can selectively wait for certain thread to finish, as far as the thread can be associated with a variable. For example:

```
x = y.domagic() & a = b.dogood() & // dispatch two threads
waitfor x;  // wait for first thread to finish
```

{ x = y.domagic(); a = b.dogood(); } &
{ c = d.doevil(); f = g.wastetime(); } &
waitfor f; // wait for the second thread to finish, the thread is identifiable by the last assignment variable

Luban engine guarantees the object integrity in the multi-threading environment. If more than one threads operating on the same object, the order of operation is not deterministic, though the integrity of each individual operation is guaranteed. Luban maintains object integrity by allowing only one thread to operate on a variable at any time of execution. Though if user builds a data type internally doing pointer reference counting, then user is responsible to make this data type thread safe. Because Luban can only see different variables and there is no way for Luban to be aware of the internal data sharing of the user data type.
One question is what will happen to those dispatched threads when the main execution thread comes to an end. By default Luban will wait for every dispatched thread to finish before doing the cleaning up. So it is like there is a hidden *wait* statement at the end of all scripts and structures. You can change this behavior by explicitly using *cancel* statement.

2.7 Tangled Threads Surprise

Take a look at this example:

for(i=0;i<10000:++i) { x += i; } &

The above seemingly innocent code will almost certain fail because it will try to start 10000 threads that will likely be more than any machine can handle. Above code will start one thread per loop. The right way to put a loop into a thread is to whole loop into a pair of brace brackets like below:

{ for(i=0;i<10000:++i) { x += i; } } &

Thread can be tricky sometimes. For example, the below simple code can produce output that surprise you.

```
for( i=1; i<=10; ++i )
{
        std::println(obj=i) &
}
```

At the first look, you would think the program print out number 1 to 10 in that order. But the truth is that the program will print out 10 numbers, but which 10 numbers and the order of the numbers are undetermined. The print could vary at each run of the program.
The root of chaos is the thread dispatching part of this code. Every print operation will be dispatched to a different thread. So ten threads will be dispatched. Together the main thread, there will be eleven threads running in this code. The execution order of these eleven threads is not deterministic.

You can use the *wait* and *waitfor* statements to control the thread execution. Plus there is another tool can be used to tame the chaos. The tool is called Asynchronous Structure. And it will be described later in chapter seven.

2.8 Everything Is Object

On handling variables and objects, the major difference between Luban and other major languages like C++ and Java, is that all objects and variables in Luban are passing by value. There is no reference or pointer type in Luban. When assigning one variable to another in Luban, the object is semantically copied instead of passing a reference like in Java. Actually Luban's value semantics is very similar to C, except Luban does not have the pointer type. For example:

```
a = [1,2,3];
b = a;
b = b + [4,5]; // b = [1,2,3,4,5] and a = [1,2,3]
```

The reason behind the design of Luban's value semantics is simplicity. Value semantics is straightforward and intuitive, requiring almost zero learning to understand.

3 Introduction to Luban Component Structure

The least flexible component of any system is the user.
- Lowell Jay Arthur

This chapter introduces the basics of Luban component model. The foundation stone of Luban component model is the Luban structure, which is indicated by a key word *struct* in Luban. The below will show how to define and use a Luban structure.

3.1 Define a Luban Structure

Suppose our hello-world program becomes so popular that we want to reuse it as a component. The way to do it is to define the hello-world code as a Luban structure, as below example shows:

```
namespace demo;

struct helloworld( )
as process
{
        std::println(obj="Hello, world!");
}
```

Now let's save the above code as file "hello.lbn". Then we can write the below script to call this component.

```
demo::helloworld(=);
```

We then save this script as "callhello.lbn". And finally we can run this new componentized hello-world program by typing the following command in the command window of your operating system:

```
Mycomputer > luban hello.lbn callhello.lbn
Hello, world!
```

Let's take a look at this new hello-world program in details. In "hello.lbn" file, we define a Luban structure named "helloworld". Luban requires every component must reside in a name space. That is why we see the line "*namespace demo;*" at the top. So the full name of this component is "*demo::helloworld*". The key words "*as process"* indicate the body of the Luban structure is a process. The difference between a process structure and other structure will be discussed in later chapters. All we need to know now is that we can use all the constructions used for scripting in the body of a process structure.
In the next script "callhello.lbn". We make a call to the component by using its full name. The syntax "*(=)"* indicates a dynamic call to a Luban structure. The details of the dynamic call to a Luban structure can be found in later chapter and here we only need to know the call will invoke the execution of the code in the Luban structure "*demo::helloworld*".

One thing need to be mentioned here is that in the above example Luban source code, there is no referring to the actual source code file name, component is only referred by its full name. This is no incidental, it is purposely designed so to avoid hard coding source code media. Yet we don't care about this for now.

3.2 Hello World Structure with Input and Output

The previous example defines a Luban component structure without any input and output properties. In practice, a component as a processing unit may need input and output properties to pass information, as the below example shows:

```
namespace demo;

struct greeting
(
        input:
                saywhat, towhom;
        output:
                greetingmsg;
)
as process
{
        msg = input.saywhat + ", " + input.towhom + '!';
        std::console().writeline(msg);
        output.greetingmsg = msg;
}
```

The above Luban code defines a Luban structure with two input properties, *saywhat* and *towhom*. The internal process of the structure combine the two input property values into a single greeting message and set it to output property named *greetingmsg*. The side effect of this structure is to print out the greeting message to the console. As we can see, inside the structure, "input.property" is the convention used to refer input and output properties.

As you can see from the example, a Luban structure definition has two parts. First is the structure interface definition, which is defined by listing the input and output properties in between the parenthesis after the structure name. The second part is the internal implementation of the structure. In this the internal of the structure is defined as a process with code between the brackets.

Now let's save this Luban component code as "helloio.lbn". Then we need to write a script to use this new component, as below:

```
demo::greeting(saywhat="Hello", towhom="world");
```

Save the above one line in a file named "callhelloio.lbn", then run the following command on your operating system like below:

```
Mycomputer > luban helloio.lbn callhelloio.lbn
Hello, world!
```

From the example, arguments are passed into structure to be called by using *propertyname=value* pairs in parenthesis. The syntax difference between a Luban structure call and regular function call in C is that instead of lining up the arguments by order, arguments are explicitly paired with input properties by name.

3.3 More On Use of Luban Structure

So far the previous examples show the use of Luban structure in a way just like calling functions. But there is a key difference between C function and Luban structure. Luban structure is object with state indicated by its property values, while C function is mostly stateless. Luban structure is more flexible and can be used in many different ways, as below example shows:

greeter = demo::greeting(saywhat="", towhom=""); // call component
greeter.saywhat = "Hello"; // print Hello,!
greeter.towhom = "world"; // print Hello, world!
greeter(saywhat="Nice to meet you"); // print Nice to meet you, world!
greeter(=); // print Hello, world!
greeter.(saywhat = "How are you", towhom="Luban"); // print How are you,Luban!
greeter.(=); // print How are you,Luban!
msg = greeter.greetingmsg; // read component object output property
std::println(obj=msg);// print How are you, Luban!

Save this file as "callhelloalt.lbn", then run the script as below:

```
Mycomputer > luban helloio.lbn callhelloalt.lbn
,!
Hello,!
Hello,world!
Nice to meet you,world!
Hello,world!
How are you, Luban!
How are you, Luban!
How are you, Luban!
```

Here we explain some details of the script line by line. The first line construct a Luban structure, assign to a variable, while triggering the execution of the structure once in the process. That is why we see a ",!" output line above. The next line set the value "Hello" into the input property named "saywhat" of the structure in variable "greeter" and the setting of input trigger the execution of the structure. That's why we see the second output line "Hello,!". Similar process happens for the third line of code, which sets the property "towhom" to value "world". The fourth line is a structure call to a structure variable instead of to a structure type name as we previously know. And this call overrides the property "saywhat" with value "Nice to meet you", making the fourth output line "Nice to meet you, world!".

It is important to know that the call to a structure does NOT change the state of the structure at all. Even with the explicit overriding of property values in the call, the property values of the original structure remain the same during the call. Actually the

structure call generates a temporarily new structure that can be assigned to a different variable. And that explains the fifth line of code calling the original structure without arguments ends up printing the same greeting as before.
As mentioned above, property setting will trigger the execution of the structure. Sometimes we may want to set multiple properties while we only want one code execution after we finish. That is why we have the sixth line, which setting multiple properties of the structure to different values, and executing the code once the settings of properties finish. And that is why we see the sixth output line "How are you, Luban!". Just to show the setting of multiple properties does change the state of the structure, the seventh line set the structure property again though this time without actual arguments. It still triggers one evaluation and that's why we see the "How are you, Luban" output once again. The last line of the script read the output property "greetingmsg" of the structure and print it to console. That's the third "How are you, Luban!" from.

3.4 Pass By Value Semantics

It is necessary to once again iterate the pass by value semantics of Luban. The passing of arguments in Luban structure call is by value, plus it is a one way street. What happens inside the structure has no possible impact on the caller. For example, the following code will result in syntax error:

```
namespace demo;

struct NotWorkingSwapObj
(
        input:
                x,y;
)
as process
{
        temp = input.x;
        input.x = input.y; // this line has syntax error
        input.y = temp; // this line has syntax error too
}
```

The reason for the syntax error is that Luban does NOT allow assignment toward input property inside a structure. Inside the structure input properties can only be read.

3.5 Name Space Basics And Program Entry Point

There are two kinds of Luban source code unit (i.e file) that Luban interpreter accepts. One is Luban script, which is a sequence of Luban process statements and expressions. Luban interpreter will simply execute the script upon request.
Another kind of source code unit is component structure definition module that defines Luban structures. The first line of a component definition module must be the definition of name space in which the following structures will reside. Luban name space uses a syntax similar to C++ name space. When Luban interpreter encounter a component definition module, it will scan the module and register the structure components into a

global name space data structure so it can be used in later call. Luban script can be used as the entry point of an application or system, while Luban component definition module can be used like library.

4 Structure Interface and Inheritance

Different towns, different people,
Somehow they're all the same.
- Long Time Gone, Bob Dylan

Component oriented programming can be compared with LEGO toys in many aspects. First you need to put the parts into the right boxes so you can easily find them out when you need them. That is what Luban name space does: put components into organized places. Second you need to recognize the parts by their shape and the way they make connections so you can use them in the right place. And that is what Luban structure interface and inheritance do: formalizing the contract with external world by defining interface, and reflecting the similarity among different components by inheritance. The goal of Luban name space, structure interface and inheritance is the same: to keep our component workshop nice and neat.

4.1 Structure Interface: Property Flow Type and External Access Permission

As shown by the examples in previous chapter, the interface of Luban structure consists of a list of different kind of properties. The previous examples have shown the basic usage of *input* and *output* properties. And below we describe all different kind of properties and their behavior in a little formal way.

There are four different kinds of flow types for properties in Luban: *input, output, store* and *static.* And the properties can have different read/write permission for external depending on their flow type. There are three kind of permissions: *readwrite, readonly* and *hidden*, and using these key words can only change the access permission for external caller. From within the structure the access permission for properties are predefined and unchangeable. The purpose of this design is to define a general framework of information flow for Luban components. The below table shows the details:

input: For external, *readwrite*. For internal, read only. Both unchangeable.
Change of value triggers evaluation of the structure.

output: For external, *readonly.* For internal, write only. Both unchangeable.
Value can only be changed from internal by evaluation.

store: For external, *hidden* by default, changeable to *readwrite* or *readonly*
For internal, both read and write.
Change of value does not trigger evaluation.

static: For external, *readwrite* by default, changeable to *hidden* or *readonly*
For internal, both read and write.
Change of value does not trigger evaluation.
Accessible only through structure type name.

From the above table and previous examples, we can see that for a Luban structure data flows from *input* to *output*. And the flow can no way be reversed. This flow model is as fundamental for Luban as function semantics for C.

We have known *input* and *output* properties from examples. The below example shows the use of *store* and *static* properties:

```
namespace demo;

struct aggregator
(
        input   incoming;
        output  total;
        store   sofar=0;
)
as process
{
        store.sofar = store.sofar + input.incoming;
        output.total = store.sofar;
}

struct useagecounter
(
        static readonly totaleval=0;
)
as process
{
        static.totaleval++;
}
```

Let's save above code into a file "storenstatic.lbn". Then we can test our new structures with the below script:

```
// testing aggregator
agg = demo::aggregator();
for(i=1; i<=100;i++) agg.incoming = i;
std::console().writeline(agg.total);

// test static
counter1 = demo::useagecounter();
counter2 = demo::useagecounter();
counter1(=); counter2(=);
std::console().writeline(demo::useagecounter.totaleval);
```

Then we save the above script as file "testsns.lbn". Afterwards we run the test by typing the following in the command window:

```
Mycomputer > luban storenstatic.lbn testsns.lbn
5050
2
```

The structure “aggregator” adds all numbers setting to its “incoming” property together and output the total to “total” property. The store property “sofar” in aggregator is used for internal state keeping. Because Luban prohibits reading value back from output properties, *store* property is used when some state data need to be remembered among different evaluations. Store property is similar to private data of C++ class.
The structure “useagecounter” maintains a global static property “totaleval” to count the number of evaluations of the structure. The static property is actually associated with structure type instead of structure instance. From outside the structure, the static property can only be referred by <structure type name> .<static property name>. From within the structure, it can be accessed using internal property naming convention that is <flow type>.<property name>.

4.2 Structure Interface Inheritance

The interface is the service contract of structure with the external world. We have seen the property flow types and external access permission key words, which are basic terms that can be used to draft such a contract.
The structure interface declares what each individual structure type does. Though when the number of different structure type grows, a large collection of unrelated structure interfaces are still difficult to remember and manage. And that is where the Luban structure interface inheritance come to help.
Similar to the class inheritance concept in Object-Oriented programming, we want to take the advantage of commonalities among different components by categorizing them using inheritance. The idea is that it is easier to understand ten animal kind than to understand one hundred individual strange animals. The below example shows how to code inheritance in Luban:

```
namespace demo;

struct DualOp
(
        input:
                op1,op2;
        output:
                result;
)
as process
{
        output.result = error(“No actually operation for general dual operand”);
}

struct Adder implements ::DualOp
(
)
as process
{
        output.result = input.op1 + input.op2;
```

```
}
```

The above Luban code defines two structures. Structure "DualOp" is a general interface for two operand operations. Structure "Adder" inherits its interface from "DualOp" and implements the internal to add two inputs together and set to output.
You may notice the syntax "::DualOp", which is a short name for "demo::DualOp". All Luban's structure type names are full name. The short name can be used if the name space of the referred Luban type is the same as the name space of the caller component.
From this example, we can see that "Adder" inherits all input and output properties from "DualOp" and does not define any new property of its own. And we need to mention here that ONLY *input* and *output* property can be inherited. The *store* and *static* property can NOT be inherited. The reason for this design is to make interface inheritance simple.

4.3 Luban Inheritance Is Only For Interface

Unlike the general Object-Oriented languages like Java and C++, Luban's inheritance mechanism is only for the interface. When one Luban structure inherits from another, only the interface of the parent structure is taken, and that excludes *store* and *static* properties. If you wish to use the internal implementation of the parent structure, you need to express you intention explicitly, as below example shows:

```
// this example uses the Adder structure defined in previous example
namespace demo;

struct AdderWithUseCount implements ::Adder
(
        static usecount=0;
)
as process
{
        static.usecount++;
        output = ::Adder(input=::Adder.input).output;
}
```

The above example defines a structure "AdderWithUseCount" that works the same as the structure "Adder", only with one more static usage counter property to count the number of evaluations.
The magic in this example is the last line of the structure process code. This line simply says: take the portion of input inherited from "Adder" structure, feed to a call to structure "Adder", then take the output of this call and set to the output of structure "AdderWithUseCount".
What actually happens in this example is very much like AdderWithUseCount inherits both interface and implementation from Adder. Though the taking of "Adder" structure's implementation is explicitly stated in one line of code. The purpose of this design is to avoid the tricky ambiguity issue for inheritance by allowing programmer to state intention explicitly. Luban allows multiple inheritance of structure interfaces.

4.4 Abstract Structure

Similar to interface concept of Java and pure virtual class of C++, Luban has abstract structure to represent pure structure interface. Abstract structure can be used for inheritance and type checking. We can redo our *"Adder"* example using abstract structure like below.

```
namespace demo;

abstract struct DualOp
(
        input:
                op1,op2;
        output:
                result;
);

struct Adder implements ::DualOp
(
)
as process
{
        output.result = input.op1 + input.op2;
}
```

In above sample code, we define an abstract structure named *demo::DualOp* and *demo::Adder* inherits its interface and implements actual operation. The inheritance relationship can be checked with type checking expression, like below Luban code shows:

```
adder = demo::Adder();
test = adder isa demo::DualOp;
```

The value of the variable *test* in above example is Boolean value *true*. The details of type checking expression can be found in next chapter.

Abstract structure can only be used for inheritance and type checking. Construction of abstract structure not allowed. For example, below Luban expression results in error value.

```
errvalue = demo::DualOp();
```

4.5 No Cyclic Inheritance

Cyclic inheritance is not allowed in Luban. Not only direct cyclic inheritance among structure interfaces is not allowed. All structure type names used inside structure interface definition including names used in property data type definition and property value initialization expression are considered part of its dependent symbols. Luban requires all its dependent symbols (interface) resolved before the structure interface can

be resolved itself. Thus cyclic reference among dependent symbols will also result in error. For example, below Luban structure definitions are invalid because of cyclic reference of structure type symbols.

```
namespace demo;

struct TweedleDee
(
        input:
                x = ::TweedleDum();
        output:
                y;
)
as process
{
        output.y = 1.0;
}

struct TweedleDum
(
        input:
                a = ::TweedleDee();
        output:
                b;
)
as process
{
        output.b = 1.0;
}
```

The above two structure “TweedleDee” and “TweedleDum”, though they don’t directly inherit from each other, they refer to each other in their property initialization expressions. And that is also cyclic interface definition and it is invalid.

4.6 Stationary Structure

Stationary Structure in Luban is used purely to hold data. It can be used almost the same way as a generic map data object. But the difference with Stationary Structure and map is that Stationary Structure is typed (so type checking works for it) and it is much faster than generic map data type. It is a good practice to define always bundled data stream as Stationary Structure to make it standard, similar to data bus concept in hardware design. Below sample code shows how to use Stationary Structure.

```
namespace demo;

stationary struct GlobalStation
(
        static hostname, cpuinfo, osinfo;
```

```
);

stationary struct DataPair
(
        name, value;
);
```

Above code defines two Stationary Structure, one to hold static global information, another one to collect two always paired data item into a single structure.
By default the properties defined in Stationary Structure always have their access permission as *readwrite* unless otherwise explicitly specified.
Stationary Structure can be inherited, serialized and de-serialized the same way as regular structures.

5 The Dynamic Type System of Luban

Life is a box of chocolate,
You never know what you're gonna get
- Forest Gump

As we all agree with Forest Gump about life's unpredictability, we may cope with it by checking each chocolate before we take a bite. And that is exactly what Luban's dynamic type system does. An object can be anything, yet you can check what it is if you want to.

All the objects inside Luban can be separated into two categories: data types and structure types. Structure types are the components coded in Luban. You can create as many kinds of Luban structure types as you wish just like you can code as many functions as you wish in C.
Data types are those built-in and/or imported data types with predefined set of member functions and applicable operators. Map, set, vector, double, int, bool, char and string are the examples of built-in data types, while std::file, net::socket are examples of imported C++ data types. The fundamental difference between structure type and data type is that you can not code a new data type with Luban. One design philosophy of Luban is that the basic data types are low level software elements that should be accessible from Luban, but Luban should not be used to make low level elements. Just like bricks are used to build house, instead of bigger bricks. Luban's belief is there should be a fundamental programming paradigm shift from software element level to component level.
Though structure types and data types are naturally different. They still obey a more or less common set of rules. These rules are to be discussed in this chapter.

5.1 Name of Data Type and Structure Type

The naming rule of Luban data type and structure type are the same. They all must be a full name with name space. But Luban built-in types are keywords and do not need name space prefix. For example:

```
string;  // built-in type
std::file; // standard imported data type
net::listener;  // imported data type
net::socket; // imported data type
std::print; // a structure type
```

The name space part in a type name can be omitted only when referred in a component that resides in the same name space, as below code shows:

```
namespace std;

struct print
(
        input obj;
)
```

```
as process
{
        ::console().write(input.obj);
}
```

The above code simply wraps the calling of member function of *std::console* type in a structure named *std::print*. We can see the referring to *std::console* in *std::print* structure does not use the name space part.

5.2 Construction of Data Type and Structure Type

The syntax of object construction in Luban is type name followed by parenthesis, as shown in the below example:

```
// print hello world
console = std::console();
console.writeline("Hello, world!");

// print hello world again
printer = std::print();
printer.obj = "Hello, world!\n";
```

Parameters can be used in the object construction. But the syntax of argument passing is different for data type and structure type. Look at the below example:

```
// create file as data type
file = std::file("helloworld", 'w');
file.writeline("Hello, world!");

// print hello world to console
printer = std::print(obj = "Hello, world!\n");
```

To pass parameters into data type construction, you just need to line the parameter values up. To construct a structure with specified property values, you need to pair up the property names with its values. Another big difference for structure construction with specified property values is that if the specified properties include input property, the structure will be evaluated once after construction. Yet the structure construction without any specified property values does not trigger any evaluation.

5.3 Explicit Type Casting

Type casting in Luban is just an object construction with a single argument. Like below:

```
int("100"); // result 100
string(100); // result "100"
double(100); // result 100.0
int(3.1415); // result 3
```

Casting to string type is special because it is universally enabled for all Luban built-in and imported data types. Actually when *std::console* type is asked to print an object, it will cast the object to a string then print the string.

Type casting does not apply to structure type.

5.4 Dynamic Type Checking: *isa* and *typeof* Operators

You can dynamically check the type of any object in Luban with *isa* and *typeof* operators, as shown in below Luban script:

```
d = 3.1416;
s = std::file("pie", 'w');

//  check type and decide what to do
if ( d isa double and s isa std::file )
        s.writeline(d);
else
        s.writeline("The end of world is coming");

// do the same thing in a different way
if ( typeof(d) == double and typeof(s) == std::file )
        s.writeline(d);
else
        s.writeline("The end of world is coming");
```

The structure interface inheritance is taken into account for type checking. For example the variable "test" in below Luban code has value *true* because structure *demo::Adder* inherits interface from *demo::DualOp* (Previous example):

```
adder = demo::Adder();
test = adder isa demo::DualOp; // test = true
```

5.5 Type Expression

One unique feature of Luban is type expression, which gives more power and flexibility to Luban's type system. Type expression is part of the Luban expression. The result of type expression can be assigned to variable and passed around just like normal object. The details of type expression can be shown in the following Luban code examples:

Type name alone can be a valid type expression.

```
stype = string; // type name is type expression
if ( not "Hello"  isa dtype ) std::console().writeline("Insane");
```

Multi-type type expression is a collection of different type expressions. The *isa* operator returns true if the type of object matches one of the types in a multi-type.

```
streamtype = < std::file, net::socket >
s1 = std::file("destfile",'w');
```

s2 = net::socket("destserver", 6500);
if (s1 isa streamtype and s2 isa streamtype) std::println(obj = "Good");

Enumerated type is a collection of constants. The *isa* operator returns true if the value of object equals any one of the collection. Look at the below example:

littleprime = < 1,2,3,5,7 >;
test = 5 isa littleprime; // test = true

The type name for type object is *typeinfo.* Below shows how it works:

s*tringtype = string;*
test = stringtype isa typeinfo; // test = true

5.6 Gate Keeper: Typed Property and Variable

There are tools in Luban to enforce strong typing policy if necessary. The type of property can be explicitly declared with type expression in structure interface definition to make sure only the objects of specified type can pass through. And also local variable can be bound to certain type using type expression. The setting of object of wrong type to a typed variable will result in the abortion of structure evaluation or script execution.

We can see the use of typed variable from below Luban code:

double dx;
dx = "Hello"; // evaluation aborts here
std::console().writeline("You should never see me, the world should have ended before me");

The above example assigns a string value to a variable "dx" that is a variable with predefined type *double.* The assignment results in the aborting of the evaluation of the script. So the last line of code should never be executed and the print should never appear on screen.

The below example shows the use of typed property in structure interface:

```
namespace demo;

struct powertoint
(
        input:
                < double, int > base;
                int power;
        output:
                result;
)
```

```
as process
{
        result = 1;
        for( I = 0; I < input.power; I++)
                result *= base;
        output.result = result;
}
```

We can save the above code to a file "powerint.lbn". Then we can write another script to call this new component, as below

```
sqr = demo::powertoint( base = 4, power = 2 ).result; // pwr = 4*4
std::println(obj=sqr);
sqrt = demo::powertoint( base = 4, power = 0.5 ).result;
if ( sqrt isa error ) std::println(obj="Good, to get square root failed as expected");

pwrcalc = demo::powertoint();
pwrcalc.power = 0.5; // evaluation stops here
std::println(obj="Surprise! World should have ended before me");
```

We then save the above script as "testpowerint.lbn" and start the test like the following:

```
Mycomputer > luban powerint.lbn testpowerint.lbn
16
Good, to get square root failed as expected
```

We can see the first evaluation of *demo::powertoint* succeeded and gave result 16 to print. The second structure call having property "power" set to 0.5 that violates the *int* type of the property results in an error value, causing the printing of "square root failed" message. The line before the last line sets a double value into the property "power" of *int* type, causing the evaluation to stop. That's why we don't see the last line get executed.

5.7 Typedef to Save You Some Typing

You can use *typedef* to define a symbol that is just the alias of another symbol. You can save some typing or create a layer of encapsulation in this way, as below code shows.

```
namespace demo;

typedef std::print print;
```

Now you have a symbol "demo::print" that is an alias to the real type "std::print".
You can also use typedef to define commonly used type expression as a symbol, like below:

```
typedef <std::file, net::socket, std::console> media;
```

Now you have a type symbol "media" that represents one of the three types.

6 Component Composition

Poor boy, pickin' up sticks
Build ya a house out of mortar and bricks
- Po' Boy, Bob Dylan

So far we have learnt how to script Luban and write Luban structure component as process code. As we mentioned before, there is another way to define a Luban structure, which is composition. The way composition defines a Luban structure is to lay out the sub-components and specify the dependencies among them. The evaluation mechanism of composition cell is to propagate the data change. The Luban structure defined as composition has the same interface as process structure and can be used the same way. In this chapter, we go through the component composition with a series of examples.

6.1 Composition 101

The basic format of Luban composition is simply a list of sub-components. The sub-component in a composition is called component cell. Each component cell has a name and a definition. The definition of the component cell specifies the content of the component cell and its dependencies with other component cells outside. The below example illustrate a basic composition:

```
namespace demo;

struct simplecomp ( output oneoone; )
as composition
{
        A1: 100;
        A2: A1+1;
        output.oneoone: A2;
}
```

The above composition computes 100+1 in a way like spreadsheet. There are three component cells in above structure. A1 contains a number 100. A2's content is an expression "A1+1". The last one with a special name links the result of A2 cell with output property "oneoone". Then we use the below Luban script to call this structure:

```
result = demo::simplecomp(=).oneoone; // result = 101
```

To show the difference between composition and process, we change the structure "simplecomp" to below:

```
namespace demo;

struct simplecomp ( output oneoone; )
as composition
{
```

```
        output.oneoone: A2;
        A2: A1+1;
        A1: 100;
}
```

The structure “simplecomp” behaves exactly the same as before. In Luban composition, the order of listing of the component cells does not matter, as far as all the cells and their dependencies remain the same, unlike the process structure for which the order of operations is essential.

6.2 Different Component Cells

There are two kinds of component cells that can be used in a Luban composition, ad hoc cell and typed cell. We discuss these two kinds of cells in this section.

6.2.1 Ad hoc Cell

Ad hoc cell contents are very much like what people normally put into spreadsheet cells. They can be constants, expressions. In Luban composition, the ad hoc cell can also contain Luban script code to do more sophisticated processing, as shown below:

```
namespace demo;

struct compadhoc
(
        input    in1, in2;
        output  out;
)
as composition
{
        A1: input.in1 + 1;
        A2: input.in2 + 1;
        A3: { std::println(obj=“run A3”); A3=A1*A2; }
        A4: { std::println(obj=“run A4”); A4=A2*A2; }
        output.out: A3+A4;
}
```

The above composition uses only ad hoc cells. The cell A1 and A2 are simple expressions. Just as expected, Luban will evaluate the expression and save the result in the cell.

Cell A3 and A4 contains Luban script code. When data updating event reaches these two cells, Luban execute the script code just like normal script. The difference between an expression cell and a script cell is that script cell does not get any value returned from script by default unless the script in the cell explicitly set the value of the cell itself, like cell A3 and A4 do in above code.

The last line in above example links the result of expression “A3+A4” to the output property “out”. The output property linking cell is one special kind of cell that can only contains single expression, no script code is allowed.

6.2.2 Typed Cell

Typed cell is to enable user to directly put pre-defined structure component into composition, like below example:

```
namespace demo;

struct TwoAdder
(
        input op1, op2;
        output result;
)
as process
{
        output.result = input.op1+input.op2;
}

struct FourAdder
(
        input op1,op2,op3,op4;
        output result;
)
as composition
{
        adder1:  demo::TwoAdder(op1=input.op1, op2=input.op2);
        adder2: demo::TwoAdder(op1=input.op3, op2=input.op4);
        adder3: demo::TwoAdder(op1=adder1.result, op2=adder2.result);
        output.result: adder3.result;
}
```

The above example illustrates how to compose an adder that can add up four numbers from adder that adds two. You can see inside the "FourAdder" composition there are three "TwoAdder" components inside, two of them linking to input properties while the third one adds the result of first two and send result to output property.

Actually the typed cell is not much different from expression cell, except that the typed cell can not be used by its own value like ad hoc cell. The other cells of the same composition can only use the output property of the typed cell.

6.3 The Execution Order of Composition Cells

The major difference between Luban Composition Structure and Procedure Structure is that composition only defines the dependencies among cells instead of execution order. The cells will be executed at run time, triggered by data updating events, while the cells to be executed and their order will be decided at run time. More details will be explained in this section through examples.

Here we use the slightly modified sample code of previous demo::compadhoc.

```
namespace demo;

struct compadhoc
(
        input    in1, in2;
        output  out;
)
as composition
{
        A1:{ std::println(obj="run A1"); A1= input.in1 + 1; }
        A2: { std::println(obj="run A2"); A2= input.in2 + 1; }
        A3: { std::println(obj="run A3"); A3=A1*A1; }
        A4: { std::println(obj="run A4"); A4=A2*A2; }
        output.out: A3+A4;
}
```

Let's save above code into a file named "compadhoc.lbn". Then we continue to write a Luban script to call this component demo::compadhoc as below.

```
 x = demo::compadhoc(in1=1, in2=2); // line 1
std::println(obj=x.out);    // line 2
std::println(obj="change input in1"); // line3
x.in1=0; // line 4
std::println(obj=x.out); // line 5
std::println(obj="change input in2"); // line 6
y =x(in2=1); // line 7
std::println(obj=y.out); // line 8
```

Save above script into a file named "start.lbn", then we can start running the program as below.

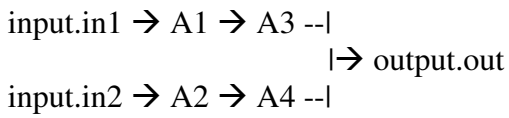
```
Mycomputer > luban compadhoc.lbn start.lbn
run A1
run A2
run A3
run A4
13
change input in1
run A1
run A3
10
change input in2
run A2
run A4
5
```

Now let's go through the program and its output line by line.

- In start.lbn the code line 1 calls Luban component demo::compadhoc, it also set input "in1" to 1 and input "in2" to 2 when calling. The resulting new component instance of the call is assigned to variable x.. Luban component demo::compadhoc runs when it is called, and it prints out the below along the way.

```
run A1
run A2
run A3
run A4
```

We now check why demo::compadhoc prints out above output when it runs. Inside demo::compadhoc the data flow is defined as below.



According to above data flow dependencies, it is required that cell A1 to be run before A3, and A2 before A4. So the output "`run A1 run A2run A3 run A4`" is in compliance with the defined data flow dependencies.

There is one very important point that needs to be mentioned here is that: for those cells that have no direct or indirect dependencies between them, the order of execution is UNDEFINED. In another word, Luban interpreter has the freedom to run them in any order or even in parallel. Luban programmer should not take any assumption about their execution order.

For our above example, there is no dependency between cell A1 and A2, neither between A3 and A4. So between cell A1 and A2, which one will be executed first is undefined. So are cell A3 and A4. In this example Luban interpreter runs A1 before A2, and A3 before A4. But this order is INCIDENTAL instead of defined behavior of Luban language. In another word, if you implement a different Luban interpreter. It runs this program and prints out "`run A2 run A1 run A4 run A3`". The behavior is still correct according to Luban language definition.

The Luban component demo::compadhoc does the computation (in1+1)*(in1+1) + (in2+1)*(in2+1) and set result to output "out". So at line 3 it prints out the output as 13 for input in1=1 and in2=2.

- The line 4 of start.lbn changes the input "in1" of Luban component instance "x" to value 0, thus triggers the evaluation of the Luban component. Here we are going to show the biggest difference between Luban procedure component and Luban composition component. For Luban procedure component any change to its input property triggers the execution of the whole procedure. For Luban composition component, change of one input property value triggers the execution of ONLY those cells that are at the data flow downstream of the changed property. In another word, if a cell inside the composition component does not have any dependency on the changed property, it will NOT be executed. Luban composition component can be "partially" evaluated depending on the changed property.

In our example, we know the downstream cells of input property “in1” are A1 and A3. Only A1 and A3 are evaluated for the change of property “in1”. That is the reason of the output:

```
run A1
run A3
```

After the evaluation of A1 and A3, the output property “out” is changed to value 10. So at line 5 it prints out 10 when it prints out the output of the component instance x.

- Line 7 of start.lbn dynamically calls the Luban component instance “x”. It also sets the input property “in1” to value 1 when calling. The mechanism of cell execution for dynamic call is the same as the setting of input property value for Luban composition component. So Luban duplicates the component and then only execute the cells that are at the downstream of changed input property “in2”, which are A2 and A4. Thus we have this output:

  ```
  run A2
  run A4
  ```

  The output is changed to value 5 after input “in2” is set to value 1. So the last line prints out 5.

We can draw the following conclusions from above.

1. The execution order of the cells inside a Luban composition component is determined by their data flow dependencies. The cells that do not have dependencies among them does not have defined execution order.
2. The change of one input property will only trigger the execution of those cells that are at the data flow downstream from the changed input property.

The “partial execution” feature of Luban composition component could be useful for certain kind fo applications.

6.4 Dependency Specification Rules

Luban composition gives user the freedom to put expression, script and typed structure into component cells, while Luban figures out the dependencies automatically for you. Yet there are still rules for dependency specification to make sure the composition is semantically clear and unambiguous.

6.4.1 Change Nothing But Yourself, But Do Read Your Surrounding

In a composition, a cell can not modify other cells, it can only read from them. These operations are considered as modification: directly assignment, member function call and property setting. So if there is such operation upon other cells in a cell definition, Luban will report error. This is to make sure the data flows one way as commonly expected.
A cell can change itself by directly assign value to itself. Though a cell can not read the previous value of itself back. So calling member function on itself is also prohibited

because it involve reading previous value back. The purpose of this rule is to prevent a cell from having dependency on itself to create an infinite loop in data flow.

6.4.2 No Tail Chasing

Luban does not allow cyclic dependencies among synchronized cells in a composition. The details of synchronized cells can be found in later chapter. So far in all the sample compositions we have seen, all the cells are synchronized.

6.5 Inheritance and Composition

Composition can be used together with Luban interface inheritance to create new structure based on old one, like below example shows:

```
namespace demo;

struct AdderPlus implements TwoAdder
(
        output diff;
)
as composition
{
        output: demo::TwoAdder(input=demo::TwoAdder.input);
        output.diff: input.op2 – input.op1;
}
```

In above example, the structure “AdderPlus” takes the interface of “TwoAdder”. It also put one instance of “TwoAdder” into its composition and wired the right portion of its input to the “TwoAdder” and wired the output of “TwoAdder” directly to its output. So “AdderPlus” basically get interface and implementation of “TwoAdder” in this way. It also adds one extra output of its own, which is the difference of its two input properties. The first line in the composition can only be used with structure inheriting interface from other structure. Luban will do type checking to make sure the wiring work with inheritance. There could be ambiguity in the case of multiple inheritance. The basic rule to handle ambiguity is to explicitly specify the wiring of the output property on which the ambiguity occurs.

6.6 Composition VS Process

Composition does its best when there are significant number of components in a system and the dataflow among the components are complicated. It is difficult to code multi-directional dataflow with process that is basically one dimensional.

Process should be used when the system have sophisticated control logic, meaning if conditions and loops. In practice, it could be the case that small components more likely use process while large components could be more compositional.

7 Asynchronous Structure: Implicit Thread and Queue

Anyone who attempts to generate random numbers by deterministic means is,
of course, living in a state of sin.
- John von Neumann

So far the Luban structure we have seen can be called synchronous structure, meaning the property values of a structure reflect the state of the structure and you can expect the change of input properties will trigger the internal evaluation and likely the change of output properties to put the structure into a new consistent state. And the internal evaluation happens in an atomic fashion, meaning Luban guarantees that the structure is always in a consistent state in every access.

In this chapter we introduce a new kind of structure: asynchronous structure. Asynchronous structure can have all the input, output, store and static properties just like the synchronous structure. The major difference is the semantics of the set and get of input and output properties. For an asynchronous structure, the semantic of input and output is more loose. What is defined is that data flows into input and flows out of output, and there is no consistency pairing between input and output. And the data flows in and out a asynchronous structure will be queued, unlike the synchronous structure, the new property value will overwrite the previous one. The store and static property value of asynchronous structure will not be queued, queuing is only for input and output. Asynchronous structure behaves like pipe, while the synchronous structure is more functional.

The purpose of asynchronous structure is to model asynchronous components in system. The examples of asynchronous components include network messaging and interactive user interface. And usually large system more likely behaves in an asynchronous way.

In actual implementation of Luban, the asynchronous structure is implemented as an object with a live thread inside. The inside thread takes data from input queue and put the output to the output queue, while outside user can put new data onto input queue and take output from output queue. So asynchronous structure does multi-threading, only implicitly.

7.1 The First Asynchronous Structure Example

```
namespace demo;

asynch struct multiplexer
(
        input flow1, flow2;
        output merged;
)
as process
{
        output.merged = input.flow1 &
        output.merged = input.flow2 &
}
```

The above example defines a asynchronous structure named “multiplexer”. The structure merges data input from “flow1” and “flow2” into the output “merged” just like a telecom multiplexer. Inside the “multiplexer”, there are two user coded threads, one to read “flow1”, another to read “flow2”, both put the reading into “merged”. There are several details of asynchronous structure that make this structure work.

First, every read of input property removes one value from the input queue. When the queue is empty, the read operations will block and wait until new value comes in. Input property can not be read from outside. Remember in the case of synchronous structure, repeated reading of the same input property read back the same value.

Second, every assignment to the output property adds one value to the output queue. For the same operation, synchronous structure will only take the last assignment to the output property. Reading of output property from outside will block if the output property queue is empty.

Understanding the above details, we can code a simple case of producer and consumer in which producer and consumer share the same queue for communication.

```
namespace demo;

struct simplequeue
(
        input inq;
        output outq;
)
as process
{
        outq = inq;
}
```

The above component behaves like a simple multi-thread queue with one end to put in and another end to read out data, plus it will make reader wait when the queue is empty. Then we can use the above component in a sample script like below:

```
// ……

q = demo::simplequeue();

{ obj= myns::producer().produce(); q.inq=obj;  } &  // producer thread

{  myns::consume( obj = q.outq ); } &  // consumer thread

// ……
```

Above script starts two threads, one to produce object and put it into the “inq” of the queue named “q”, while another to read object out of “outq” of “q” and feed to “myns::consume” structure. When the “outq” is empty, the consumer thread will sleep.

Above examples show how to code asynchronous structure and use it in Luban script. The next section will show how to code and use asynchronous in composition.

7.2 Use Asynchronous Structure to Synchronize

In chapter one, when we introduce thread dispatching, we give out an example that shows some nasty surprise when threads are dispatched uncontrolled in a loop. Actually that kind of loop is commonly used in certain situation. For example you could write a server that watch a network port and dispatch thread to handle received message. And this server is simple a forever loop with thread dispatching inside, like below code shows.

```
while ( true )
{
        x = port.getmsg();
        ::msghandler(msg=x) &
}
```

Above code is problematic, because the execution order of the main thread and dispatched thread is not deterministic. It could happen that the main loop goes faster that the message "x" is changed before the "::msghandler" thread read it. The way to fix this is to use Asynchronous Structure, as below code.

```
handler = ::asynchmsghandler();
while ( true )
{
        x = port.getmsg();
        handler.msg=x;
}
```

And you define your "asynchmsghandler" like below:

```
asynch struct asynchmsghandler(input msg;)
as process
{
        // handle msg;
}
```

Since the object "handler" contains an Asynchronous Structure that has a thread inside. And all inputs set to "handler" will be correctly queued to be processed in order.

7.3 What Happens to Asynchronous Structure When Script Ends

In chapter one, we described that at the end of script execution, the main thread will wait for all explicitly dispatched threads to finish. Luban engine does not give any special treatment to the live thread inside Asynchronous Structure, meaning it just destroy it like anything else. And the destruction of Asynchronous Structure will stop the internal thread even if it is in the middle of running. The below scripts show the semantics.

```
asynchprinter = demo::AsynchPrinter();
for( i=1;  i<11; ++i)
        asynchprinter.obj = i;
```

And demo::AsynchPrinter is defined as below.

```
namespace demo;

asynch struct AsynchPrinter(input obj;)
as process
{
        std::println(obj = input.obj);
}
```

It may surprise some user that the above script may print nothing, instead of a complete 1 to 10 integer list. The reason is that the main thread destroys the "asynchprinter" objet before it can finish its internal thread can finish its job. Modify the script to use the "waitfinish" member function of Asynchronous Structure will fix the problem.

```
asynchprinter = demo::AsynchPrinter();
for( i=0;  i<10; ++i)
        asynchprinter.obj = i;
asynchprinter.waitfinish();
```

The *waitfinish* member function will only return when the input queue of the structure is empty and there is at lease one thread inside sleeping and waiting for new input. For our example, it will guarantee that all objects from 1 to 10 are printed before the script quits.

7.4 The Life Cycle of Asynchronous Structure

We already know that you can trigger the evaluation of synchronous structure by setting its input property and/or by explicitly making structure call. And we expect the evaluation to finish before the execution flow goes further down.

Asynchronous Structure is different by nature since it has a live thread inside. The following describes the basic behavior of Asynchronous Structure.

- When Asynchronous Structure is created, its internal thread is inactive.
  ```
  x = demo::asynchinput(); // no thread started here
  ```
- Setting the input property or getting its output property will trigger the internal thread to start running.
  ```
  y = x.newinput;  // this will trigger the internal thread running in x
  ```
- You can manually force internal thread to start by calling the *start()* member function.
  ```
  x.start(); // this will start the internal thread of x too
  ```
- If an Asynchronous Structure has inputs, its internal thread always loops back to wait for new inputs automatically. It will stop otherwise.
- When an Asynchronous Structure is out of scope, its destructor will be called and it will force the internal thread to stop, even if it is still active.

- User can call the *waitfinish()* member function to wait for the internal thread of an Asynchronous Structure to become inactive. Inactive internal thread either sleeps waiting for input or completely stops.

  x.waitfinish(); // this will block until internal thread of x becomes inactive
- The assignment of one Asynchronous to a new variable will give the new variable a inactive structure with empty input and output queue. In another word, the input and output queue are copied.

  y = x; // y will also be an asynch struct like, but no i/o queue and inactive
- Asynchronous Structure does not support structure call. Making structure call to Asynchronous Structure will generate error value.

  y = x(input1 = 0.0); // y will be of error value because x is aynch struct

7.5 Asynchronous Structure in Composition

Asynchronous structure and composition naturally work together. While in process script, user need to pay attention to the semantic difference of an asynchronous structure, in composition, user wire the asynchronous the same way as synchronous structure. Luban take care of the independent thread inside the asynchronous structure automatically.
In below example, we code one asynchronous structure to listen to network message. Then we use the structure in a composition.

```
namespace demo;

asynch struct messenger
(
        output msg;
)
as process
{
        socket = net::socket("msgserver", 6500);
        while ( true )
        {
                obj = socket.readobj();
                if ( obj == "good bye" ) break;
                output.msg = obj;
        }
}

struct listenNprint
()
as composition
{
        MSG:  ::messenger();
        PRINTER: std::println(obj = MSG.msg );
}
```

Above code creates an asynchronous structure to listen to the port 6500 on server "msgserver", and set the output "msg" whenever object arrives from the server. The "messenger" structure will only stops when it hears "good bye" from the server. The next composition uses this structure as its component and wire it together with a structure that prints message on console. We can run the composition structure using the below script:

```
demo::listenNprint(=);
```

When the composition "listenNprint" runs, it will listen to the message on network print what it hear on the console until the server says "good bye" to it.

7.6 Ad Hoc Asynchronous Cell In Composition

You can put ad hoc asynchronous cell into a composition by simple declare the ad hoc process cell as "asynch". The code inside the cell will follow the asynchronous structure semantics automatically. We can code the "listenNprint" structure using ad hoc asynchronous cell as below:

```
namesapce demo;

struct listenNprint
()
as composition
{
        asynch MSG:  { socket = net::socket("msgserver", 6500);
                       while ( true )
                          {
                                obj = socket.readobj();
                                if ( obj == "good bye" ) break;
                                MSG = obj;
                          }
                       }
        PRINTER: std::println(obj = MSG );
}
```

You can notice that in the ad hoc cell, there is no output property to set except itself. And its fellow component cells can only refer the value of itself instead of the output property.

7.7 Asynchronous Component and Cycle In Composition

As mentioned in previous chapter, cycle consists of only synchronous cells in a composition is not allowed. In another word, if the cycle has one or more asynchronous component inside, it is OK for Luban. Let's look at the below example:

```
namesapce demo;

asynch struct player
(
```

```
        input receive;
        output send;
)
{
        output.send = 1; // send a ball
        while ( true )
        {
                ball = input.receive;
                std::println(obj=ball);
                if ( ball == 1000 ) break;
                ball++;
                output.send = ball;
        }
}

struct pingpong
()
as composition
{
        P1:  ::player(receive = P2.send );
        P2: ::player(receive = P1.send );
}
```

It is obvious that the composition structure “pingpong” has a cycle involving cell P1 and P2. Yet Luban is still going to happily take it and run through because P1 and P2 are asynchronous component, so it does not violate Luban’s no-cycle rule.
If you run “pingping” structure, it will print out number 1 to 1000, and print each number twice in a row.

8 Error Handling of Luban

The web of our life is of a mingled yarn, good and ill together.
- William Shakespeare

Luban use a special type: *error* type to handle errors. The value of error type is generated as error happens in Luban code evaluation. The basic principle of Luban error handling is very simple and can be described in below two points:

- Propagate error value along the line of expression, assignment and control flow if sensible.
- Abort the evaluation of Luban structure or script if the error can not be propagated in a sensible way.

8.1 So Long Try and Catch Madness

There are enough "who catch what at where" confusion about exception in many different languages. Luban goes a different way, it does not have exception at all, neither try and catch statements. The basic idea is that to use error value to represent error is simpler and easier to understand. Error value also fits Luban's data flow model very well.

8.2 Error Value Generation and Propagation

```
// below variables all contain error

firsterr = 1/0;
alsoerr1 = firsterr++;
alsoerr2 = firsterr + alsoerr1;
test = firsterr isa error and alsoerr1 isa error and alsoerr2 isa error; // test = true
```

In above script, first error is generated by operation to divide with zero. Then the error propagates through a series of expression and assignment statements. So the error happens and spread logically corresponding to the code structure. Responsible user can decide to check error and take action at any point.

8.3 Errors That Stop The Show

There are certain points that is error sensitive in Luban. When errors happen at those points, Luban can only stop the evaluation of the current structure or script and report error. These points are listed as following.

Failure of assignment is one of the error sensitive points. Failure of assignment can be caused by assign object of wrong type to typed variable or structure property, for example:

```
int intvar;
intvar = 3.1416; // world ends here, assign double to a int type variable
std::println(obj="Devil shall never see the light"); // never reach this line
```

If the result of condition expression inside control flow statement is not of *bool* type, Luban will stop the evaluation because it can not determine the control flow.

```
X = 0;
if ( X )   // X is not bool type, evaluation stop
  X++;
```

When the call to a process structure stops because of error, the call returns error value instead of a fully evaluated structure.

Except the above error sensitive points, when error happens Luban will happily continue and pass the error object around. It is totally up to the user to decide where they want to check the error or if they want to check the error at all.

9 Utility Data Types and Structures

Men have become the tools of their tools.
- Henry David Thoreau

There are a set of commonly used data types and structures packaged together with Luban programming language. Though they are not part of the Luban programming language, they are so useful that we need to use a chapter to describe them.

9.1 Console, File and Socket

These are the basic data types used to persist information or pass them around network. And they share a more or less common interface. Below listed are their Luban type names and the description of their member functions.

Luban type names.

Console:	std::console
File:	std::file
Socket:	net::socket

Member functions that apply to all above data types:

write(obj1,obj2…):

Writes the object in human readable string format to the media.
Return null if successful, or error type in case of failure.

writeline(obj1,obj2…):

Same as *write*, only adds one extra line break at the end of each object string.
Return null if successful, or error in case of failure.

read(int n=1):

Reads specified number of characters, or read one character if *n* not specified.
Returns a string containing all characters read, or error in case of failure.

readline():

Reads the media to break of line.
Returns a string, or error in case of failure.

writeobj(obj1,obj2…):

Writes the objects in platform independent binary format from which the object can be restored.
Returns null if successful, or error for failure.

readobj():

Restore object back from media, likely the object written by *writeobj.*
Returns restored object.

What need to be mentioned is that *writeobj* and *readobj* are designed to be used in pair. Any object serialized and persisted with *writeobj* can be restored using *readobj*.

9.2 How To Create, Read and Write File

Below is a simple text file processing example:

```
linecount = 0;
```

```
wordcount = 0;
txtfile = std::file("mydatafile", 'r'); // open file "mydatafile"
alldata = txtfile.readall(); // read ALL content of the file into alldata
lines = alldata.split('\n'); // split alldata into vector lines
foreach( line in lines )  // Iterate through each line in the vector lines
{
        ++linecount;  //increase line count
        words = line.split();  // Split one line into a vector of words
        wordcount += words.size(); // increase word count
}
//print result
std::println(obj="lines: "+string(linecount)+" words: "+string(wordcount));
```

It needs to be mentioned that above code use a new file type member function *readall()* that reads the whole content of a file into a string data object. The code then split the whole content into lines and does analysis.

The below example script shows how to use file to persist and restore data object using file:

```
fw = std::file("objfile", 'w'); // open a file for writing
fw.writeobj([1,2,3]); // write a vector object
fw.close(); // close the file
fr = std::file("objfile", 'r'); // open file for reading
vec = fr.readobj(); // read one object back
std::println(obj=vec); // print out [1,2,3]
```

There are three modes can be used to open a file. They are read, write and append, and are represented by characters as 'r', 'w' and 'a'. Above code shows read and write. While in the write mode opening an existing file will cause it to be truncated, opening the same file in append mode will cause the later writing append to the end the file.

9.3 Socket Routine, Server and Client

The below scripts demonstrate how to use socket, including both client and server side. The server side uses a new data type *net::listener* that is basically a TCP port listener. It can be created given a port number. Its major member function is *accept()* that accepts one incoming connection and return a socket object that can talk to the client.

Server script:

```
// This server greets clients and assign each of them a number
listener = net::listener(6500); // take TCP port 6500
clientcount=0;
while( true ) // forever loop
{
        socket = listener.accept(); // wait for client connection
```

```
        name = socket.readobj(); // read client name
        socket.writeobj("Hello, "+name); // say hello
        ++clientcount; // increase client number
        Socket.writeobj(clientcount); // send the number to client
}
```

Client script:

```
socket = net::socket("localhost", 6500); // connect to server at port 6500
socket.writeobj("Chinese");  // send server its name
msg = socket.readobj(); // receive greeting
std::console().writeline("The server says: ", msg); // print greeting
mynumber = socket.readobj(); // read assigned number
std::console().writeline("My number is ", mynumber); // print number
```

In the above, the server script takes TCP port 6500 and listens to incoming connection. For each client connection server says hello then assign a number to the client. Client script simply connect to the server at port 6500, send its name, then receive greeting message and number. You can run both scripts on the same machine. If you want to run them on different machines, you need to change the sever name from "localhost" to the actual server host name.

9.4 Utility Structures: std::println and std::print

Just for convenience, std::println and std::print structures are coded as part of Luban package. The structure print object to console in a simple way. Below is their source code:

```
namespace std;

struct println( input obj;)
as process
{   std::console().writeline(input.obj); }

struct print( input obj;)
as process
{   std::console().write(input.obj); }
```

9.5 Utility Structure std::des

Every Luban object can dump itself into a binary stream. In order to restore one object out of a binary string, the *std::des* structure is needed, as below sample code shows.

```
x = {"one":1, "two":2};  // construct a map
xguts = x.serialize();//  serialize it into a string
xback = std::des(stream=xguts).obj; // call std::des to restore it
truth = xback == x; // truth = true, compare the restored object and original
```

10 The Luban Java Bridge: LJB

There is one utility API need to be specifically illustrated in this chapter, which is the LJB Luban Java Bridge. LJB is the interface through which Luban can construct any type of Java object, call any Java instance and static member functions, read and write Java instance and static fields. In short speaking, you can utilize everything available in Java through LJB. The significance of Java language and the richness of its packages are the reasons that I use this whole chapter to show how LJB works.

10.1 Constructing Java Object in Luban

Luban imports all Java object as an external type *java::javaobj.* The construction of Java object is the same as constructing other Luban data types. Below are examples of Java object constructions.

```
javapie = java::javaobj(“java.lang.Double”, 3.1416);  // build a java double number
javahello = java::javaobj(“java.lang.String”, “Hello, World!”); //  java string
javanow = java::javaobj(“java.util.Date”); // obtain current date/time from java
```

From above you can see the way to construct Java object is actually very simple, all you need to do is to always put the Java class name as the first argument and put the arguments the class constructor needs afterwards.

10.2 Calling Java Object Member Functions

Calling Java object member function using LJB is exactly the same as calling any object member function in Luban or Java. The below are examples.

```
jmap = java::javaobj("java.util.HashMap");
jmap.put("one",1);
one = jmap.get(“one”);
std::println(obj=one);
```

The above code constructs a Java hash map object and calls the “put” member function to add one key value pair into the map. Then the following code then reads out the value by looking up by the same keys with “get” member function. The last line print out the look up result to show.

You can see it is extremely simple to call Java object member function in Luban, just construct the object and call as you would do in Java itself.

10.3 Calling Static Member Function

To call a Java static member function is also simple, like below code shows.

```
jdumy = java::javaobj();
connection = jdumy.java_callStaticFunction(
"java.sql.DriverManager","getConnection",URL, user, pswd);
```

The way to call Java static function is to go through a special member function of Luban java::javaobj type. The Luban java::javaobj member function name is

“java_callStaticFuncion” and its arguments are Java class name, static function name followed by actual arguments for that Java static function. Because the calling Java static function does not require any specific Java object instance, we can use a Java null object as a gateway to call any Java static functions.
In the above example, we first construct a Java null object using the default constructor of java::javaobj type. The we use the “java_call StaticFunction” to call the Java static function “java.sql.DriverManager.getConnection(URL, user, pswd)”. This Java static function is to obtain a connection to a SQL server.
Below is another example of calling Java static function. This example is to split a string using Java regular expression utilities.

```
jnull = java::javaobj();
pattern = jnull.java_callStaticFunction("java.util.regex.Pattern", "compile", "[,\\s]+");
jresult = pattern.split("one,two, three,  four,   five"); // jresult is java String[]
```

The above code calls Java static function “java.util.regex.Pattern.complile()” to construct a Java regular expression pattern, then uses the pattern to split a string into an array of Java strings.

10.4 Access Java Static Fields

The way to access Java static data fields is similar to call its static member function, you need to use two special member function on the java::javaobj object, which can be Java null since the field is static. The two functions are “java_getStaticField” for reading and “java_setStaticField” for writing. Take a look at below code example.

```
dumy = java::javaobj();
hourfield = dumy.java_getStaticField("java.util.Calendar", "SUNDAY");
std::println(obj=hourfield);

status = dumy.java_setStaticField("java.util.Calendar", "SUNDAY", 2); // try to set
std::println(obj=status);  // should print error message can’t set final field
```

Above code first gets the value of a Java static field “java.util.Calendar.SUNDAY” that is of value integer 1. Then it tries to set the same field value to integer 2, which is going to fail because the field is final. The code will print out the value of the field and the error status message for the field value set operation.

10.5 Access Java Instance Fields

To access Java instance data field, you need to use a valid java::javaobj instance that has that data field accessible. It can not be a Java null object. You still use special member functions to access, and the member function names are “java_getField” for reading and “java_setField” for writing. Take a look at below example code.

```
    jdim = java::javaobj("java.awt.Dimension", 200,200);
    height = jdim.java_getField("height"); // read height field
    std::println(obj=height);
```

```
jdim.java_setField("width",100); // set width to 100
std::println(obj=jdim);
```

The above code constructs a Java object of Java type “java.awt.Dimension” that has two accessible fields “height” and “width” that are initialized to be both of value 200. Then the code reads the value of “height” field and prints it out. The following code then sets the “width” field to value 100 and print out the whole object to show the effect.

10.6 Checking Java Object Type

You can easily do “instanceof” checking on a Java object in Java. The similar thing you can do in Luban is “isa” operation. Though for Luban all Java objects are of type “java::javaobj” and you can not check the object’s Java type using Luban ”isa” operator. The way to check the actual type of a Java object in Luban is to call another special member function “java_instanceof”. Below is one code example.

```
x=java::javaobj("java.lang.Double",3.1416);
xisajavadouble = x.java_instanceof("java.lang.Double");
std::println(obj=xisajavadouble);

numberclass = x.java_getClassForName("java.lang.Number");
xisajavanumber = x.java_instanceof(numberclass);
std::println(obj=xisajavanumber);
```

In the above example, we first construct a Java object that is of Java type “java.lang.Double”. Then we call the “java_instanceof” function to check if the object is of type “java.lang.Double”. The function call should return a Java Boolean value true and we print it out. In the following code we do the similar type checking, yet using a different argument to call the “java_instanceof” function. This time we use a Java class object as the argument for checking. And the Java class object for Java class “java.lang.Number” is obtained from another Luban java::javaobj member function “java_getClassForName”.

“java_getClassForName” is a function to load and return any Java class object using its class name. When class object loading is a frequent operation, this function becomes handly.

10.7 Java Object Is, Actually, the Java Object Reference

I have explained how to construct Java object, call its static or instance member function, access its static or instance data field plus how to do Java’s “instanceof” type checking in Luban. Everything seems smooth so far.

Though there is one important thing I haven’t explained. In Java all objects are object references, while in Luban all objects are values. How do we bridge the difference?

The solution is technical. In Luban Java Bridge, the java::javaobj type holds the real Java object reference, while from Luban side, the VALUE of the object of java::javaobj is actually the REFERENCE of that real Java object that it is holding. Luban treats java::javaobj like C treats its pointer type.

So in another word, Luban has no choice but to honor Java's reference semantics because that's the only thing we can get from Java. The below code shows the impact of Java object reference in Luban code.

```
jvec1 = java::javaobj("java.util.Vector");
jvec2 = jvec1;
jvec1.add(1);
std::println(obj=jvec1);
std::println(obj=jvec2);
std::println(obj= jvec2==jvec2 );
```

Run above code will print out two identical Java vector both contain element 1, and print out the result of testing "jvec1==jvec2" as Boolean value true. Since java::javaobj type contains only Java object reference as its value, so the assignment to a new variable jvec2 does NOT actually generate a new copy of the Java Vector object. The two variables contain the reference to the same actual Java object. So the print statements print out the same vector and the equality checking ends with value true.
In order to get a new copy, you need to call the clone() member function (if available), like what you would do in Java itself. Below modified code operates on two different instances after calling the clone() function.

```
jvec1 = java::javaobj("java.util.Vector");
jvec2 = jvec1.clone();
jvec1.add(1);
std::println(obj=jvec1);
std::println(obj=jvec2);
std::println(obj= jvec2==jvec2 );
```

Actually the object reference kinds of data types already exist in Luban before Java object. The Luban types like std::console, std::file and net::socket are all using reference in their implementations. So the assignment of file, console and/or socket to a new variable does not end up with new copy, the reference gets copied instead of the real object.

10.8 Using Java GUI Widgets and Luban InvocationQueue Class

One of the most interesting uses of Luban Java Bridge is to use Java Swing widgets to do GUI programming. With all the facilities offered by Luban Java Bridge, it is easy to construct Java GUI widget. With a little extra help from a special Java class called Luban.InvocationQueue, we'll be able to propagate the GUI event back into Luban's domain, and the whole thing will then just work. Below is a Luban GUI code example:

```
frm = java::javaobj("javax.swing.JFrame", "Hello Swinger");
jpnl = java::javaobj("javax.swing.JPanel");
jbut = java::javaobj("javax.swing.JButton", "Push me");
jpnl.add(jbut);
frm.setContentPane(jpnl);
```

```
frm.setSize(java::javaobj("java.awt.Dimension", 300,100));

queue = java::javaobj("luban.InvocationQueue"); // new  invocation queue, line 7

listener1 = queue.newProxy("java.awt.event.ActionListener"); // line 8
jbut.addActionListener(listener1); // line 9

listener2 = queue.newProxy("java.awt.event.WindowListener"); //line 10
frm.addWindowListener(listener2); // line 11

// Run below code between brackets in a different thread
{
 for(i=0;  ; ++i)
 {
        //below gets the next event in the queue, block if queue empty
        event = queue.getInvocation();

        std::println(obj=event);
        if ( string(event.getMethod().getName()) == "windowClosing"  ) break;
        jbut.setText("Push Me "+string(i));
 }
}&

frm.setVisible(true);
```

Run the above code and you will see a Java Swing window with a button inside labeled "Push Me". When you click the button the label of the button will change to "Push Me X" with X to be the number of times the button has been pushed. And in the console, when button is pushed the event message will be printed.

Now let's go through the details of this program. The first five lines of the program do the simple widget construction. It constructs a JFrame, a JPanel and a JButton. It then put the JButton into the JPanel, JPanel into the JFrame and set the size of the JFrame to 300 by 100. All these are very much like what you would do in Java, just in a slightly different syntax.

The code afterward takes care of the GUI event call back. Here we need to introduce the Java class Luban.InvocationQueue. This class is used here as the gateway to pass Java GUI event into Luban. The two most important member functions of this class are "newProxy" and "getInvocation". Luban.InvocationQueue.newProxy() member function takes the name of a Java Interface and generates a Java proxy project to mimic the Java Interface. The generated Java proxy object can be used any Java function that takes that Java Interface. When the member function of that Java Interface gets invoked, the proxy object handles it in a very simple manner, it puts the member function method object and its arguments into a invocation queue that resides inside the original Luban.InvocationQueue.

The luban.InvocationQueue.getInvocation() function simply fetch one invocation from the invocation queue. The function blocks and waits when the queue is empty. The next

new invocation will then wake up the function and it will get the new invocation and continue.

Now we can get back to how Luban code uses the Luban.InvocationQueue class to handle the GUI events. In our example, line 7 creates the InvocationQueue object. Then line 8 calls the “newProxy” member function to create a Java proxy object with the Java interface “java.awt.event.ActionListener”. Line 9 uses that Java proxy object as the call back listener for the JButton object we created before. According to my explanation above, every time the button get pressed, the member function “actionPerformed” of the Java Interface “java.awt.event.ActionListener” will be called and the Java proxy object will put the “actionPerformed” method object and the argument into the invocation queue. Line 10 creates another proxy object with Java interface “java.awt.event.WindowListener”. Line 11 uses that proxy object as the listener of the JFrame object to listen to window events like window closing. Please note you can create multiple proxy objects on the same queue. That means all the invocation upon all the different proxy objects will end up in the same queue.

The following Luban code handling the GUI event is dispatched into a separate thread. Within the thread, it is an infinite loop. In the loop, the code first takes one event out of the invocation queue. If the queue is empty it will block and wait until next invocation comes. Then the code prints out the invocation event details. Afterwards the code check if the event is a window closing event, and break out of the loop and finish the thread if it is. Remember we have the windows event and JButton event in the same queue. Otherwise it changes the label of the JButton to include the number of loops and continue.

It is a good practice to put the event handling of one invocation queue into a separated thread so the main thread can continue when it is waiting for the next invocation event. The next example uses two queues and two threads to do pretty much the same thing as above example. Below is the code.

```
frm = java::javaobj("javax.swing.JFrame", "Hello Swinger");
jpnl = java::javaobj("javax.swing.JPanel");
jbut = java::javaobj("javax.swing.JButton", "Push me");
jpnl.add(jbut);
frm.setContentPane(jpnl);
frm.setSize(java::javaobj("java.awt.Dimension", 300,100));

queue = java::javaobj("luban.InvocationQueue");
q1=queue;
listener = queue.newProxy("java.awt.event.ActionListener");
jbut.addActionListener(listener);

queue2 = java::javaobj("luban.InvocationQueue");
listener2 = queue2.newProxy("java.awt.event.WindowListener");
frm.addWindowListener(listener2);

{
 for(i=1; ; ++i)
```

```
// thread to handle the button click event
 {
        event = queue.getInvocation();
        if ( event == null ) break;
        std::println(obj=event);
        if ( string(event.getMethod().getName()) == "windowClosing"  ) break;
        jbut.setText("Push Me "+string(i));
 }
}&

// a different thread to handle window events
{
 while(true)
 {
        event2 = queue2.getInvocation();
        std::println(obj=event2);
        if ( string(event2.getMethod().getName()) == "windowClosing"  )
        {
                q1.close();  // close another queue
                break;
        }
 }
}&

frm.setVisible(true);
```

When the above example is run, it does the same thing as previous example does. The difference between this example and previous example is that previous example use one invocation queue to handle the events from both the JButton and the JFrame widgets, while this example use two different queues for JButton and JFrame. And this example uses two different threads to obtain and handle the events from that two different event queues. In this example, when one thread detects that window is closing, it closes the other queue that handles the button events, so the other thread can return from "getInvocation" call with null and finish.

One trick, when you call the InvocationQueue.close() function of another queue, you need to use different variable that contains the reference to the queue used in another thread, like in above example we use variable "q1" instead of the "queue1" variable. The reason is that Luban has a thread safe feature that only allows one function call to one variable at any time. So while the "queue1" variable blocks at "getInvocation" function call, the "queue1.close()" can not get through and that will prevent both thread from finishing. So we assign "q1=queue1" first, and later we use q1 to close the queue. Because all Java objects are actually object references, this trick works.

The luban.InvocationQueue and associated luban.InvocationEvent are included in luban.jar file that is part of the Luban source and binary package.

10.9 JDBC and Regular Expression Examples

Talking to a database is commonly required for applications. Here I give one example for using JDBC interface to talk to a MySQL database server.

```
javadumy = java::javaobj();
drvclass = javadumy.java_getClassForName("com.mysql.jdbc.Driver");
std::println(obj="load jdbc driver:\t"+string(drvclass));

URL = "jdbc:mysql://mypc:3306/mydb";
user = "root";
pswd = "rootpassword";

conn = javadumy.java_callStaticFunction( "java.sql.DriverManager",
"getConnection",URL, user, pswd);
std::println(obj="get connection:\t"+string(conn));
st = conn.createStatement();
rs = st.executeQuery ("select * from stocks");

std::println(obj="Content of stocks table:");
while ( bool(rs.next()) )
{
        stockname = rs.getString(1);
        stockprice=rs.getDouble(2);
       std::println ( obj=string(stockname)+"\t"+string(stockprice));
}
rs.close();
st.close();
conn.close();
```

The above example is actually very similar to any Java JDBC example code. They go through the same routine. First it loads the MySQL jdbc driver class by calling the Luban java::javaobj utility member function "getClassForName()" on a dummy java::javaobj instance. Then it calls the Java static function java.sql.DriverManager.getConnection to obtain a connection to the MySQL server with specified URL, user name and password. Then it calls the "createStatement()" member function on the connection object, then execute a query. The query is to get the content of a table named "stocks". After it gets the results for the query, it then iterates through the results and print out contents. The program nicely closes everything at the end.

Regular expression is another commonly used tool for text processing. Below example shows how to use Java regular expression tools in Luban through LJB.

```
// EXAMPLE 1, string split
jnull = java::javaobj();
pattern = jnull.java_callStaticFunction("java.util.regex.Pattern", "compile", "[,\\s]+");
strtosplit = "one,two, three,,  four,   five";
jresult = pattern.split(strtosplit); // jresult is java String[]
```

```
result=vector(jresult); // convert String[] to luban vector
std::println(obj=strtosplit);
std::println(obj=result);

// EXAMPLE 2, find and replace
pattern2 = jnull.java_callStaticFunction("java.util.regex.Pattern", "compile", "cat");
original = "one cat, two cats in the yard";
matcher = pattern2.matcher(original);
match = bool(matcher.find());
jbuffer = java::javaobj("java.lang.StringBuffer");
while(match)
{
        matcher.appendReplacement(jbuffer, "dog");
        match = bool(matcher.find());
}
matcher.appendTail(jbuffer);
result2 = string(jbuffer);
std::println(obj=original);
std::println(obj=result2);
```

The above Luban code example uses Java regular expression classes to do two things, one is to split a string another is to find all “cat” words in a string and replace them with word “dog”. In the first example, it constructs a Java regular expression pattern by calling a Java static function. Then it calls the “split” member function of the pattern to split a string. Finally it converts the Java string array object to a Luban vector and prints the vector out. In the second example, it also constructs a Java regular expression pattern. Then it builds a matcher for the string to be processed with the pattern. It then goes looping to find all the occurrences of the pattern “cat” and replace it with “dog” and put the result into a Java string buffer. After the loop is done, it converts the Java string buffer to a Luban string and prints it out.

10.10 Luban and Java Data Type Conversion

There are two kinds of conversions we will discuss below, Luban to Java and vice versa.

10.10.1 Luban to Java Conversion

The Luban to Java conversion is always implicit. It happens when Luban data type is used to construct Java object and call Java function. For example, when below Luban code runs:

```
Jdouble = java::javaobj(“java.lang.Double”, 3.1416);
```

The constant 3.1416 is a Luban data type, it will be converted to a Java Double type and used as one argument to construct a Java object that happens to be also a Java Double type. The conversion is implicit because the code does not state to do that conversion explicitly.

The rules for Luban to Java type conversion is listed below.

Luban Type	Java Type

double	java.lang.Double
int	java.lang.Integer
bool	java.lang.Boolean
char	java.lang.Character
string	java.lang.String
map	java.util.HashMap
vector	java.util.Vector
set	java.util.HashSet
java::javaobj	NO CONVERSION NEEDED

One more note, the Luban to Java conversion is always a "deep" conversion. It means that if a Luban container type is converted, not only the container will be converted, every element inside the container will be converted recursively.

10.10.2 Java to Luban Conversion

Differing from Luban to Java conversion, Java to Luban conversion has to be explicit, or it will not happen. What I mean here is that any the Java object that Luban obtained from JVM with Java constructor or function calls, it will be always wrapped as a special Luban data type "java::javaobj" you can use this object just like any other Luban object without conversion.

There are situations you want to cast Luban Java object into some other Luban type. Then you need to use Luban cast expression like below:

```
jvec = java::javaobj("java.util.Vector");
jvec.add(1.0); // use it directly
lubanvec = vector(jvec) // cast to Luban vector;
foreach( x in lubanvec )
        std::println(obj=x);
```

The above code converts a Java vector object into a Luban vector using a cast expression, then use foreach statement on the converted Luban vector. If no conversion happens you can still use the Java object inside variable "jvec" as it is. The above code casts the Java vector into a Luban vector only to iterate through it.

All Java to Luban conversions have to be explicit with a cast expression. Below list all Luban types that be used as cast operators and the Java types applicable to them.

Java Types	Luban Type
All array types + java.util.List	vector
java.util.Map	map
java.util.Set	set
java.lang.Boolean	bool

java.lang.Number + java.lang.Character	char
java.lang.Number	double
java.lang.Number	int

Also different from Luban to Java conversion, Java to Luban conversion is always shallow, meaning when converting a Java container type to another Luban container type, only the container itself get converted, while all the elements inside container kept untouched.

10.11 Build, Run and JVM Parameters

To build and run LJB, there are some configurations need to be done. The below are the details.

- set up environment variable JAVA_HOME before build and install
  For example, you have your JDK at C:\jdk1.5.0_02 you need to specify this in POSIX format on cygwin:

  JAVA_HOME=/cygdrive/c/jdk1.5.0_02

  On linux, you need to do the same thing.
  If you don't have JAVA_HOME env variable setup, the build script will ask you for the value of JAVA_HOME at the beginning.
  On Windows with Cygwin, you also need to make sure you have the write permission to JAVA_HOME/lib directory. Luban build script will copy one file jvm.dll.a file into that directory to enable gcc to link against jvm.dll

- Make sure jvm.dll accessible at run time
  On Linux, for example you have Java installed on /usr/lib/java, you need to have the environment variable LD_LIBRARY_PATH setup like below:

  LD_LIBRARY_PATH=
  /usr/lib/java/jre/lib/i386/client:/usr/lib/java/jre/lib/i386

  On Windows, for example your Java installed at C:\jdk1.5.0, you need to put this into PATH environment variable:

  PATH=C:\jdk1.5.0\jre\bin\client

Luban uses two environment variables to pass parameters it needs to start the JVM, including the place to find the jar files.

The first environment variable is “CLASSPATH”. This is the variable to contain the path to all the jar files that contains the classes that’s not a part of the JVM standard distribution. For example if you have a MySQL JDBC driver package plus the Luban InvocationQueue class to load, you can put the jar file location into CLASSPATH like below:

CLASSPATH=
D:\mysqljdbc\mysql-connector-java-3.1.10\mysql-connector-java-3.1.10-bin.jar;
D:\luban\javacode\luban.jar

If you have multiple jar files and/or multiple class directories, you can list them all in CLASSPATH, separate them with semicolon.
The other environment variable is “LUBAN_JVMOPTIONS”, you can put arbitrary JVM options there as far as you separate them with space. For example, you can specify JVM maximum and minimum heap size like below:

LUBAN_JVMOPTIONS=”-Xms64m –Xmx256m”

11 Add New Data Type To Luban

The head learns new things,
but the heart forever practices old experiences.
- Henry Ward Beecher

This chapter demonstrates a way to code a C++ class and export it to Luban. This chapter is only for hard core professional who wishes to export C++ classes into Luban.
Basically the adding of new data type can be done in two steps. First is to code the class in a Luban compliant way, second is to let Luban know where to load the class.

11.1 Code a Luban Data Type In C++

Here the example is to code a new Luban data type *demo::greeter*. This data type does the simple job to print out "Hello, world" upon request.
To code any Luban compliant data type in C++, the convention is to always use two source files, one header file and one source module. In our simple greeter case, we will code header file *greeter.hpp* and *greeter.cpp*.

Below is the source code in *greeter.hpp*:

```
#include <lbtypes/lbobject.hpp>
#include <lbtypes/LBDeclareMacro.hpp>

class Greeter : public Luban::LBObject
{
        LBDECLARE( Greeter )

        public:

        string toString() const;   // for human readable printing
        ostream& toStream(ostream& o) const;  // serialize guts
        istream& fromStream(istream& i, int major=-1, int minor=-1); // restore
        virtual bool equals(const LBObject& another) const;

        // member function visible in Luban
        LBObject *luban_greet(const LBVarArgs *args);
};
```

What the header file tells us:

- All Luban data types are derived from Luban::LBObject data type.
- A mysterious macro "*LBDECLARE()*" needs to be put in.
- There are four functions must be implemented for all Luban data types.
  "*toString()*" is used to convert the data object to a human readable string, and this function makes all Luban types printable and castable to string type.

"*toStream()*" is used to serialize the internal content of the data object. This function makes Luban types universally serializeable.
"*fromStream()*" is used to restore content from a serialized data stream which is produced by the *"toStream()"* function in the same class. Please note that "*toStream()*" only needs to be responsible to "*fromStream()*" function, meaning that the stream from "*toStream()*" can be used by *"fromStream()"* to restore to the original state.
"equal" function offers the comparison among data objects in the same class or different classes.
The conclusion, a Luban data type is required to know minimum three things:

1. Print itself
2. Serialize/de-serialize itself
3. Check equality against others

- The last member function *"luban_greet"* bears a signature that is required for any luban class member function to be exported to Luban language.

Then next step is to define these in a CPP module. Below is the source code in *greeter.cpp*:

```
#include "greeter.hpp"
#include <lbtypes/LBDefineMacro.hpp>

LBDEFINE_EXPORT(Greeter, "demo::greeter", 1, 0 )

LBDEFAULT_STATIC_CONSTRUCTOR(Greeter);

string Greeter::toString() const
{
  return "Greeter";
}
ostream& Greeter::toStream(ostream& ost) const
{
  return ost;
}

istream& Greeter::fromStream(istream& ist, int major, int minor)
{
  return ist;
}

bool Greeter::equals(const LBObject& another) const
{
        return dynamic_cast<const Greeter*>(&another) ;
}

LBEXPORT_MEMBER_FUNC(Greeter::luban_greet,"greet", "string greet()")
```

```
LBObject *Greeter::luban_greet(const LBVarArgs *args)
{
        if ( args == 0 || args->size() == 0 )
                return new LBString("Hello, World");
        return new LBError("demo::greeter() function takes no arguments");
}
```

For the *greeter.cpp* module, the first thing needs to be put in is the *LBDEFINE_EXPORT* macro. From its appearance it indicates the mapping from C++ class name to Luban data type name plus the version number, and the version number can be used by the *fromSteam()* member function to decide what to do when restoring a data object stream of different version.
The line after the *LBDEFINE* macro is another macro *LBDEFAULT_STATIC_CONSTRUCTOR* that defines a default form of static constructor that does nothing more than calling the default constructor. More details of static constructor can be found in next section.
greeter.cpp also defines the mandatory virtual functions. *toString()* function returns a simple string to indicate itself as a greeter object. *ToStream, fromStream* functions do virtually nothing because there are no internal data to be serialized or restored. *Equals()* function check if the object compared against is a *Greeter* type, and returns true if it is.
The most interesting function here is the *luban_greet*. Luban requires that the member functions to be exported to Luban to have the same signature, which *luban_greet* bears. Basically the function signature can pass any number of arguments of any type, though the member function itself has to check the number and type of arguments then either perform requested action or return error value.
To export a member function to Luban, the mapping from the C++ member function name to the Luban member function name needs to be indiciated. That is what the *LBEXPORT_MEMBER_FUNC* macro does. The first argument is the C++ member function name, then corresponding name in Luban. The last argument is the synopsis of the function.

After the coding, we can compile our new class into a shared library. In our case we named the library *libgreeter.so*. That finishes all we need to do for C++ coding of a new Luban data type.

11.2 Inform Luban About The New Type

Next thing we need to do is to tell Luban how to load this new data type. For this step all we need to do is to list the new data type's C++ class name, its Luban type name and the shared library name in a file, then tell Luban to read the file at starting. Luban has a dynamic data type loading mechanism that will load the shared library when the data type is needed at run time.
For our new *demo::greeter* type, we need to put the below line into a file.

```
Greeter         libgreeter.so    demo::greeter
```

The format of the line is C++ class name, shared library name then Luban type name. Let's write the line into a file named *"import_types"*.

Now everything is ready, we can test to use this new type in Luban, like below lines show:

```
Mycomputer > luban –t import_types –i
➢  s
g = demo::greeter();
msg=g.greet();
^D
Hello, World
>quit
MyComputer>
```

In the above example, Luban interpreter is started with –t flag that tells Luban to read a file of imported types. In our case there is only one data type *demo::greeter* in the file. At Luban interpreter command line, we use "s" command to start scripting. Then Luban runs the script and print out the result of the last evaluation that is the famous "Hello, World" message.

11.3 Basic Operations

For this new *demo::greeter* type, only a minimal basic operations can be performed. These operations include assignment, equality comparison, casting to string, serialize to stream, restore from stream and member function call. The below sample code shows the basic operations.

```
X = y;          // assignment
X == y;         // equality check
x.y();          // member function call
string(x);      // cast to string
```

There is a set of standard member functions available for all Luban data types:

```
H = x.hash();  // hash key for the object, default value 0
Sz = x.size();  // apply to container and string type, default value error
x.contains(y); // apply to string and container type
x.clear();      // apply to string and container
guts = x.serialize(); // returns the binary guts stream in a string
h = x.invoke("hash"); // call hash() in a reflective way
funmap = x.allfuncs(); //  map containing all member functions and descriptions
obj = std::des(stream=guts).obj;  // restore object from a string
```

11.4 More Operations

One obvious way to expand available operation is to add and export more member functions. There is also a set of virtual functions that can be implemented so more Luban expression operators can be applied on the data type. These functions are optional. If they are not implemented, the corresponding operation will result in error value. The following lists the virtual functions and their corresponding operation in Luban.

```
void plus(const LBObject& x)          +, +=
void minus(const LBObject& x)         -, -=
void mul(const LBObject& x)           *, *=
void div(const LBObject& x)           /, /=
void mod(const LBObject& x)           %, %=
void neg();                           -x
void plusplus();                      ++
void minusminus();                    --
bool before(const LBObject& x)        <,>,<=,>=

//container related functions
int hash()                    //used as hash key when the object is used as key in a map
const  LBObject* index(const LBObject* idx)        //  a=b[c];
LBObject* index(const LBObject* idx)               // b[c] += 1;
bool replace(const LBObject* idx, const LBObject* value);          // a[b]=c;
LBConstIterator *getIterator();       // foreach( x in vectory )
int size();                           // number of elements
```

11.5 Static Constructor And Casting

As previously mentioned, Luban requires all data types to have a static constructor. Luban static constructor is simply the function Luban calls to construct the object upon request. Technically it is a relay between Luban and the real C++ constructors of the data type. There is a default implementation available in the format of a macro. But it does nothing more than calling the default constructor of the class.

You need to implement your own static constructor if you need to put arguments into object construction. A sample static constructor is like below:

```
// sample static constructor for a fake LBComplex type
LBObject *LBComplex::staticConstructor(const LBVarArgs *args)
{
        int numarg = args? args->numArgs():0;
        switch ( numargs )
        {
                case 0:
                        return new LBComplex();
                case 1:
const LBDouble *dbl  = dynamic_cast<const LBDouble*>(args->getArg(0));
                        if ( !dbl )
                                throw LBException("Wrong type");
                        return new LBComplex(double(*dbl));
                case 2:
const LBDouble *dbl1  = dynamic_cast<const LBDouble*>(args->getArg(0));
const LBDouble *dbl2  = dynamic_cast<const LBDouble*>(args->getArg(1));
                if ( !dbl1 || !dbl2 )
                        throw LBException("Wrong type");
```

```
            return new LBComplex(double(*dbl1), double(*dbl2));
        }

        throw LBException("Wrong number of args");
}
```

With above static constructor coded, user can construct complex type in Luban like below:

```
C1 = math::complex(); // default
C2 = math::complex(1.0); // complex with only real part
C3 = math::complex(1.0, 2.0); // both real and imaginary
```

From above example, we can see static constructor also perform the task of type casting, in the case of single argument to the constructor. The variable C2 contains a complex cast from a simple real number.
If we wish to cast a complex to a double, using only static constructor, we have to change the static constructor of double type, which we may not be able to do because double is a built-in type and the code is out of the reach of us. Luban offers an alternative way to implement casting for the situation. That is to implement the below virtual function.

```
void LBComplex::cast(LBObject *castto)
{
        LBDouble * dbl = dynamic_cast<LBDouble *>(castto);
        if ( ! dbl )
            throw LBException("Wrong type to cast to");
        *dbl = LBDouble( _real );
}
```

Above function is to cast the real part of the complex into the double number passed in. It will enable the below construction in Luban:

```
C = math::complex(1., 2.);
D = double( C );
```

There could be the case that both the static constructor and cast function can be applied to the same operation. And the rule is to choose static constructor over cast function.

11.6 Helper Classes

There are two helper classes in Luban C++ importing mechanism. One is class *Luban::LBVarArgs* that is used majorly to pass data into member functions. *Luban::LBVarArgs* is an abstract class.

```
class LBVarArgs
 {
 public:
```

```
    virtual int numArgs() const = 0;
    virtual const LBObject* getArg(int index) const = 0;
};
```

Another helper class *Luban::LBConstIterator* is also an abstract class. It is used by container data type to return iterator for Luban's *foreach* statement.

```
class LBConstIterator
 {
 public:
   virtual bool next()=0; // move forward
   virtual const LBVarArgs* getCurrentRow() = 0;
 };
```

Luban::LBConstIterator views each container as a number of rows and each row can contain one or more elements. It uses *LBVarArgs* interface to access elements in a row. The need to implement your own *LBConstIterator* derived class may arise when your implement your own specific container class and export to Luban.

11.7 Common Assumptions About Luban Data Types

Luban takes certain common assumptions about the data types imported from C++. The assumptions are actually in line with C++ common requirements for classes. They are listed as below.

- A good copy constructor
  This is mandatory. All types must have a working copy constructor.
- An assignment operator
  This operator will be called. Though you may implement one that does nothing more than throwing exception if necessary. Luban uses this operator to improve efficiency of variable assignment. But everything still works if the assignment operator fails.
- A good default constructor
  This is necessary when you use the default static constructor, or your own static constructor calls the default constructor explicitly. Also it is needed for streaming.
- A good destructor
- No worry about multi-threading (usually)
  Though Luban is a multi-threading language, it has a mechanism to ensure that only one thread calls member function of an object at any time. So the coder of the class normally does not need to worry about thread safety, unless you use some object sharing technology like reference counting. If you do sharing object among different instance of your class, then you need to do something make sure your class is thread safe. Because there will be more than one thread accessing your shared object.

12 Reflection

Now he worships at an altar of a stagnant pool
And when he sees his reflection, he's fulfilled.
- License to Kill, Bob Dylan

A complete dynamic reflection mechanism has been built into Luban language. All objects and components can be dynamically constructed by string name that is only determined at run time. All structure properties can be dynamically set and get with string property name. And all object member functions can be called via string name too. The below sections describe more details.

12.1 Construct Object Using Reflection

You can construct any data objects and components by their full names. The proxy for reflective object and component construction is a data type named "luban::ns" that represents the namespace of Luban. By calling the create() member function of *luban::ns* object with the name string of object and component type, you can construct objects and components dynamically, as the below sample Luban script shows.

```
// make a namespace object
ns = luban::ns();

//Basic type construction
pi = ns.create("double", 3.1415);
i = ns.create("int", 3);
b = ns.create("bool");
c = ns.create("char", 'a');
s = ns.create("string", "Hello, Reflection")
v = ns.create("vector", 10, 0);
m = ns.create("map");
s = ns.create("set");
socket = ns.create("net::socket", "localhost", 8500);

//component construction
adder = ns.create("luban::demo::simpleadder");
adder.(op1=100, op2=200);
std::println(obj="component output: "+string(adder.result));
```

From above code you can see you can call the object construction by its type name and pass in arbitrary number of arguments following the string type name. The whole mechanism is dynamic. The below code constructs a default instance of the same type as variable *x* whatever the type is.

```
xdefault = luban::ns().create(string(typeof(x));
```

12.2 Get Type Information Object Using Reflection

Not only you can create any object instance using reflection, you can also get the type information object using reflection and use that object for type checking purpose. Take a look below code.

```
ns = luban::ns();

filetype = ns.gettype("std::file");
fileobj = std::file("xfile", 'r');
truth = fileobj isa filetype; // truth = true
```

In above code, a "filetype" object is created by calling "gettype" member function of *luban::ns.* The created object is then used to check the type of a variable named "fileobj".

12.3 Set and Get Structure Property by Reflection

The component model of Luban is based on its *struct* type that is property value flow based. The Luban's reflection mechanism covered set and get of *struct* properties by reflection, as the below code shows.

```
adder =luban::demo::simpleadder();
adder.pset("op1", 1.0);
result = adder.pget("result");
std::println(obj="component output: "+string(result));

adder.pset({"op1":1.0, "op2":2.0});
result = adder.pget("result");
std::println(obj="component output: "+string(result));

adder.pset(["op1","op2], [1,2]);
result = adder.pget("result");
std::println(obj="component output: "+string(result));

inputs = adder.pget(["op1","op2"]);
std::println(obj="component inputs: "+string(inputs));
```

Above code shows that you can set and get *struct* properties by using its built-in member function *pset* and *pget*, with property name string passed in as argument. You can set multiple properties at once by passing a map with property names as keys or a vector of keys plus a vector of values. To get property values *pget* function can be used and if multiple values need to be retrieved at once a vector of property names can be used as arguments. And a vector of values is returned as result.

12.4 Call Object Member Function by Reflection

The member function of any Luban data type can be called via reflection too. As below code shows.

```
mp = {"John":1, "Paul":2, "George":3};
```

```
mp.invoke("insert", "Yoko", 4);
std::println(obj=mp);
```

The special member function *invoke* is the entrance for relective member function call, and *invoke* is available for all Luban types. In the above example, the *insert* member function of *map* type is called through *invoke* and a pair of key and value is inserted as result.
There is one related member function *allfuncs()* that is available for all types in Luban, which is to give out a complete map of all available member functions and their description in a map. Below code shows how to use the special *allfuncs()*.

```
funcmap = anyobj.allfuncs();
```

12.5 Reflection Usage Example: Remote Component Call

Using Luban's reflection mechanism and network streaming facilities, it is almost trivial to construct a client/server tool to enable remote calling of Luban components. On the client side, all need to be done is to send the component name, input property values plus the desired output property names. For the server, it needs to listen to the incoming connection get the component name, inputs and desired output names. Server then builds the component by name, set its inputs, get its output as required and send results back through the same socket. The whole system can be built with very little Luban code. That's why we can list the whole system code as in less than two pages below.

12.5.1 Remote Component Call Server Code

```
// a multi-threaded RCC server
namespace luban::demo;

struct RCCServer( store readwrite int port;)
as process
{
        rootlistener = net::listener(store.port);
        while ( true )
        {
                onesock = rootlistener.accept(); // receive incoming call
                ::HandleRCC(socket=onesock) &  // dispatch to a different thread
        }
}

// HandleRCC actually handle one call and return results
struct HandleRCC
(
        input:
                socket;
)
as process
```

```
{
        socket = input.socket;
        package = socket.readobj(); // get the whole package

        compname = package[0]; // first one is component name
        args = package[1]; // second one input properties
        resultprps = package[2]  //third one for desired outputs;

        comp = luban::ns().create(compname); // construct by reflection
        comp.pset(args);  // property setting by reflection
        result = comp.pget(resultprps); // property get by reflection

        socket.writeobj(result);
}
```

There are two *struct* are defined in above code. One is the main loop of the RCC server, all it does is to listen to the incoming connection and dispatch to the *struct HandleRCC* in a different thread then back to listening again. Thread dispatching is a common technique used in server coding to accommodate more than one request simultaneously.
In the *struct HandleRCC*, a package is retrieved from socket. And component name, inputs and desired outputs are unpacked. Then using reflection mechanism, component is built, its inputs are set and outputs are gotten. Then the results are written back to the same socket.

12.5.2 Remote Component Call Client Code

```
// Remote Component Call client code
namespace luban::demo;

struct RCCClient
(
        store readwrite int port; store readwrite string host;
        input:
                string compname;
                map     ins;
                outs;
        output  results;
)
as process
{
                onesock = net::socket(store.host, store.port);
                onesock.writeobj(  [input.compname,  input.ins,  input.outs ] );
                output.results = onesock.readobj();
}
```

The RCC client code is even simpler. All it does it to make a connection to the specified server host and port. Then client put user specified component name, inputs and desired

outputs into a package of vector format, write to the socket. The last step is simply waiting for the results to come back from server.

12.5.3 How to Run RCC Server and Client

The below line of Luban script will start the RCC server at port 9600.

```
luban::demo::RCCServer(port=9600);
```

The below several lines of code dispatch the work separated to multiple parts and send them simultaneously to the server and wait for them to finish.

```
client = luban::demo::RCCClient();
client.host = "superserver.mycompany.com";
client.port = 9600;
P1=client(compname="myns::supercalc" , inputs={"vectosum":[1,2,3,4]},
outs="sumofvec").results &
P2= client(compname="myns::supercalc", inputs={"vectosum":[5,6,7,8]},
outs="sumofvec").results &
P3 = client(compname="myns::supercalc", inputs={"vectosum":[9,10,11,12]},
outs="sumofvec").results &
waitfor P1,P2,P3;
total = P1+P2+P3;
```

Above code construct a RCC client to talk the server running on "superserver.mycompany.com" at port 9600. Through the connection, client calls the component named "myns::supercalc" to summarize numbers in a vector. It dispatches three calls in different thread to summarize three vectors simultaneously, wait for them to come back. Finally it then adds up the three results to get the grand total.

12.5.4 Possible Improvements of Sample RCC

Using reflection and networking, we build a very practical and useful RCC client/server system. There many ways to improve it in practice. One straight forward way is to run the RCC server on a virtual server that may dispatch jobs to a server cluster. Since cluster can probably handle the load balancing by itself, it is transparent to Luban user. For Luban user the unlimited power of parallel computer is now available at their finger tips.

12.6 Reflection and Parallel Computing: A Perfect Match

From above example RCC system, we can see that it is powerful to combine reflection and parallel network computing together in Luban. Luban makes a perfect environment to dispatch parallel jobs and synchronize them. With Luban, you have the infinite capacity of cheap computing power from server farm. You can attach problem of any complexity as far as the problem itself can be divided and conquered as smaller problems.

13 Luban Interpreter

Simplicity in character, in manners, in style;
in all things the supreme excellence is simplicity.
- Henry Wadsworth Longfellow

To run Luban code, we use a simple Luban interpreter named, surprise, “luban”. This luban interpreter takes Luban code in the files and runs them through.

The same Luban interpreter can also run in interactive mode. In this mode you can explore the content of Luban name space, and do interactive scripting. The things the interpreter can do are actually quite simple. Though I believe it serves its purpose.

13.1 Load and Run Luban Source Code Files

It is very simple to run Luban source code in the form of files. All you need to do is to put them on the command line when you start Luban interpreter. For example:

In file “saygood.lbn” type in below Luban code:

```
std::println(obj="World is good");
```

In file “saybad.lbn” type in below Luban code:

```
std::println(obj="World is bad");
```

At the command prompt of your operating system, type below:

```
Mycomputer > luban saygood.lbn saybad.lbn
World is good
World is bad
```

You can see that Luban interpreter takes the Luban script file and runs them in the same order of the command line.

There are different Luban source code file type than Luban script file. They are Luban component definition module files that only define Luban component structure without invoking any. You can freely mix Luban component definition code files with script files and Luban interpreter will handle them automatically. For example:

In file “saygood.lbn” type in below Luban code:

```
namespace demo;

struct SayGood()
as process
{
        std::println(obj="World is good");
```

```
}
```

In file “saybad.lbn”, type in below Luban code:

```
namespace demo;

struct SayBad()
as process
{
        std::println(obj="World is bad");
}
```

In file “start.lbn” type in below Luban code:

```
demo::SayGood(=);
demo::SayBad(=);
```

At the command prompt of your operating system, type below:

```
Mycomputer > luban saybad.lbn start.lbn saygood.lbn
World is good
World is bad
```

You get the same result as our first example. Though this time we used a different code structure. This time, saygood.lbn and saybad.lbn define two Luban component structure instead of two directly executable scripts. And the two component structures defined by saygood.lbn and saybad.lbn are invoked by the script inside start.lbn.
Luban always processes the component definition before running any script file, that’s why the order between script file and component definition file does not matter to Luban interpreter.
One more thing we want to mention is that the order of Luban component definition files doesn’t matter either. Luban read all component files into memory and handles the inter-dependency automatically so user don’t have to worry about as far as all components are presented at the command line.

13.2 Interactive Mode of Luban Interpreter

Below command will start Luban interpreter in interactive mode

```
Mycomputer > luban –i
6 imported types
script/s - to start scripting
edit/e  - to edit script using $EDITOR or vi
list/l <namespace name> - to browse name space
quit/q - to quit
Luban> _
```

Flag –i lets Luban interpreter get into interactive mode. In interactive mode, user can type in command to check the content of Luban namespace, enter Luban script and run it. For Luban Beta1.2 or higher, user can invoke external editor to edit Luban script. The following examples show how these command work.

```
Mycomputer > luban –i
> list std
   std::console    std::console
   std::deserializer       std::deserializer
   std::file       std::file
   std::print      std::print(input readwrite obj;)
   std::des                std::des(input  readwrite  string
   stream;output readonly obj;)
   std::printline  std::printline(input readwrite obj;)
   std::println    std::printline(input readwrite obj;)
```

Above example uses “list” command to list the content in Luban namespace “std”. Using “list” without specify namespace name will list all the content at the root of Luban namespace.

```
Mycomputer > luban –i
> script
   std::console().writeline(“Hello, world”);
   ESC\ENTER
   Hello, world
```

The above example users “script” command to start entering Luban script directly at command console. Pressing ESC key followed by ENTER will stop the entering of script and Luban will take the code that you have entered and run it through. In above example user entered a line of code and the running result is to print out “Hello, world” on screen.

```
Mycomputer > luban –i
> edit
```

Above command will invoke an external editor for user to edit Luban program. Luban will take what users edited in the external editor and run it through. It is probably more convenient than directly input at command console. User can specify external editor to use by defining “EDITOR” environment variable, like below example

```
Mycomputer > export EDITOR=xemacs
```

If the editor is not specified, Luban will try to use traditional UNIX editor “vi”. If failed to invoke, Luban will report editor error.
The external editor feature is only available for Linux version and Cygwin+Windows. The binary package for windows does not have this feature, unless user already has a working Cygwin installed.

13.3 Other Command Line Arguments

Below are more command line arguments that can be used to control the behavior of interpreter.

- -t <filename>
  This flag is to specify the name of the file that lists all the data types imported from other languages like C++ in the form of shared library. If this argument is not specified, Luban uses a default imported type list file, which is either /usr/lib/luban/imports or environemtn variable $LUBAN_HOME/imports. More details of data type importing can be found in chapter 10.

- -s <Luban code>
  Use this flag you can put a short line of Luban code in the command line when starting Luban interpreter. And Luban interpreter will run the code, like below example:

```
Mycomputer > luban –s “std::console().writeline(100);”
100
Mycomputer >
```

- All other arguments other than above flags will be taken as Luban source code file names. And all the non-component Luban script files will be run in the order of their command line position.

14 Sample Application: Topic Based Messaging System

The power of role model is infinite.
- Mao Zedong

Using Luban programming language and its networking and file utilities, it is very easy to write a simple topic based messaging system. Client can subscribe and publish message by topic, while server accepts client subscription, receive topic message update and broadcast to subscribing clients. The code required to implement such a messaging system is very small. And they are completely listed below.

14.1 Messaging Server Code

```
namespace luban::demo;

stationary struct MsgServiceConfig
(
  static readonly host = "localhost";
  static readonly subport = 9876;
  static readonly pubport = 9877;
  static topics = {:};
);

 struct SubService( )
as process
{
        subport = net::listener( ::MsgServiceConfig.subport );
        while ( true )
        {
                sck = subport.accept();
                topic = sck.readobj();
                std::println(obj="subscriber for "+topic);
                first = false;
                if ( not ::MsgServiceConfig.topics.contains( topic ) )
                {
                        ::MsgServiceConfig.topics[topic] = [null, []];
                        first = true;
                }
                ::MsgServiceConfig.topics[topic][1].append(sck);
                sck.writeobj("OK");
                if ( not first )
                        sck.writeobj( ::MsgServiceConfig.topics[topic][0] );
        }
}

struct PubService( )
```

```
as process
{
        pubport = net::listener( ::MsgServiceConfig.pubport );
        while ( true )
        {
                sck = pubport.accept();
                pack = sck.readobj();
                std::println(obj=pack);
                topic = pack[0];
                value = pack[1];
                sck.writeobj("OK");

                if ( not ::MsgServiceConfig.topics.contains(topic) )
                {
                        ::MsgServiceConfig.topics[topic] = [ value, [] ];
                        continue;
                }
                ::MsgServiceConfig.topics[topic][0] = value;
                foreach( sub in ::MsgServiceConfig.topics[topic][1] )
                        sub.writeobj(value);
        }
}

struct PubSubServer()
as process
{
  ::PubService(=) &
  ::SubService(=) &
}
```

The messaging server code consists of several components. The first one MsgServiceConfig is actually a shared data holder. It contains several static properties that are accessible for cross different Luban structures. The static properties save the information like server host name and port number. There are two ports open on the server, one to accept incoming subscription, another one to accept incoming topic updates. The essential data property is the one named “topics”. It is initialized as an empty map, it is to contain all the topics and their value, plus all the subscriber’s socket data. The key to the map is message topic. And the value of in the map is a two element vector. The first element is the value of the topic, while the second one is a vector containing all the subscriber’s connection sockets.

The execution logic of the server is coded into two structures. One structure “SubService” is to handle subscription request. It accepts the incoming request, look up the topic in the shared data map and put the socket into the vector associated with the topic subscribed. Another structure “PubService” accepts the message topic updating request. It takes the topic and value of the update, looks up the data map, update the value

of the topic, then send out the updated message to each of the subscriber through the saved socket vector.

14.2 Messaging Service Client Code

```
namespace luban::demo;

asynch struct Subscriber
(
  input:
          string topic;
  output:
          updates;
)
as process
{
  socket=net::socket(::MsgServiceConfig.host,::MsgServiceConfig.subport );
  socket.writeobj(input.topic);
  ack = socket.readobj();
  std::println(obj=ack);
  while ( true )
    {
          newdata = socket.readobj();
          output.updates = newdata;
    }
}

struct Publisher
(
  input:
          string topic;
          updates;
  output:
          serverack;
)
as process
{
  socket=net::socket(::MsgServiceConfig.host,::MsgServiceConfig.pubport );
  socket.writeobj([input.topic, input.updates]);
  ack = socket.readobj();
  std::println(obj=ack);
  output.serverack = ack;
}
```

```
struct SubClient
(
  input:
        string topic;
)
as composition
{
  sub: ::Subscriber(topic=input.topic);
  printer: std::println(obj=sub.updates);
}

struct PubClient()
as process
{
  for(;;)
    {
        console = std::console();
        line = console.readline();
        tandv = line.split();
        ack = ::Publisher(topic=tandv[0], updates=tandv[1]).serverack;
        console.writeline(ack);
    }
}
```

The above client code are separated into “Subscriber” and “Publisher”. The logic of both structure are very simple. Subscriber connects to the message server on specific host and port, send the topic to subscribe to, then it holds the socket and listen to any update and send received update its output property so the downstream components can process the update. You may notice that this “Subscriber” structure is *asynch*. And it is purposely design to be so. *Asynch* structure fit naturally with network messaging service because it runs in a different thread and put all its output in queue. The “Publisher” also connects to the message server though on different port. It simply sends the topic of message and the message itself in a two element vector. “Publisher” is a regular structure, instead of *asynch.*

The other two structures are for testing purpose. The “SubClient” structure is compositional, it simply connects a “Subscriber” to a “std::println” structure. So every updates the “Subscriber” gets from network will be printed out to standard output by the “std::println” structure. The “PubClient” simply read line from standard input in the format of “topic message” and call the “Publisher” structure to publish the topic and message.

14.3 Potential Improvements to Messaging System

You can see from above sample code that the messaging system is practically useful though the coding of it is very simple. This simple layer present the network communication function in a topic based format that is far simpler and easier than the socket.

One potential improvement to this simple messaging service is to expand it to a messaging/database hybrid system. Put it simply, if we persist all messages to a file system, it becomes a simple name key based database system. And things saved in it can be any Luban object. The hybrid system can be very interesting to use because it not only saves the data by name, it can broadcast the updated new data to subscribers cross the network.

15 Todos

I got a head full of ideas,
And that are drivin' me insane.
- Maggie's Farm, Bob Dylan

There are many things that can be done to fully realize or expand the power of Luban programming language and paradigm. A few of them are listed below.

- Luban Studio: A spreadsheet like IDE

This is an integrated part of Luban development environment. It is actually its IDE (Integrated Development Environment). The goal of this component is to hit home the intuitiveness and simplicity of Luban by presenting it in a GUI interface. One major advantage of this IDE is to enable Luban user to do composition in a way almost the same as spreadsheet construction. It will be immediately understandable for anyone who can use spreadsheet, even if he/she has never done programming before.
This Luban Studio may also include functionalities that are commonly used for programmers, including debugger, version control and cross network common component repository. The goal is to make Luban Studio a platform on which people build and share software components.

- GUI widget components for Windows, Linux and Java

It would be instantly gratifying for Luban programmers if they could easily construct an application with GUI within Luban's infrastructure. Luban has built-in space designed to put GUI widget, which is asynchronous structure. The goal of this component is to wrap commonly used GUI widgets as asynchronous structures and put in Luban name space, including Windows, Linux and even Java GUI widget.
As soon as the GUI widget available as Luban components, they can be easily used as any other Luban components. Imagine wiring GUI components in a composition model, it is hard to get anything more intuitive than that.

- Closure for Luban

It could be convenient to put a piece of code into an ad hoc closure with a name and run it as many times as you wish.

- Luban Java Bridge

It could be easier than Luban/C++ bridge because Java has a built-in reflection mechanism. This bridge will enable exporting Java classes and member functions into Luban, possibly including an API to let Java call Luban component too. This one is actually done in Luban Beta 2.0.

- Luban SQL Utility

It is simple to build a db.mysql data type for Luban to run SQL statement on a MySQL or any relational database server. Since Beta 2.0 Luban can access any DBMS through JDBC.