sa.engine C++ Interfaces

Stream Analyze Sweden AB

Version 1.1

`sa_CPPAPI.pdf`

The C++ API is a thin wrapper interface in C++11 upon the CAPI 2.0 in the namespace `sa`. All operations are inline definitions. The C interface CAPI is also included. For an overview of concepts, refer to the sa_CAPI 2.0 documentation.

To use the C++ API, include either `sa_client` or `sa_core`, but not both. The difference is thread safety. Including `sa_client` will use thread locking and one set of interface functions, while including `sa_core` will use a simpler and faster set of interface functions but is not safe for multi-threaded use.

Class sa::handle

All objects in sa.engine are referred to through an object handle of the C type `ohandle`. The handle is wrapped in the class `handle`, which is used as a base class for managing the life span of an `ohandle`, and is not intended for direct use. Handles owned by the `handle` class are automatically released when the `handle` object goes out of scope.

Handles can be reassigned, copied or moved and are automatically released, as necessary. Use the `h()` method to get the underlying `ohandle`, but be careful with it and do not use it to initialize new objects. Use normal assignment operations to get copies.

This class provides some common operations like printing (`print`), swapping (`swap`), testing for validity (`h.isvalid()`), checking the type of the underlying object (e. g. `isint()`), and throwing sa.engine errors (`h.error("message")`).

Class sa::object

An instance of the `object` class represents an object in sa.engine. It can be created with primitive types, which will create a corresponding object in sa.engine. For example:

```
sa::object a_string{"hello"};
sa::object a_real{3.75};
```

Creating an object will allocate space in sa.engine and copy the data there. Once the object goes out of scope, the handle is released. If the reference count on the handle reaches zero, the allocated storage in sa.engine will be automatically deallocated.

To retrieve the data from an `object`, use a `getx()` function. For example

```
double d = a_real.getd();
```

If the underlaying type of the object does not match the get function, an exception of type `sa::error` will be thrown. The available types for constructing objects and their corresponding getter functions are

C/C++ types	*sa::type_t*	*getter*
basic integer types; int, uint8_t, short etc.	kInteger	geti()
64-bit integer types; long long, uint64_t etc.	kInteger	getl()

const char *, std::string	kString	gets()
float, double	kReal	getd()
bool	n/a	getb()
class tuple	kArray	geta()
class record	kRecord	getr()
bin<T>	kBinary	getbin()

Binary types

For binary data, there are two flavors available:

```
bin<T> getbin<T>()          // creating a dynamic binary blob
```

This version will dynamically allocate a smart pointer (`bin<T>`) that is automatically destroyed when it goes out of scope, and can be accessed just like a pointer to T or an array of T, depending on the type T. It cannot be copied, like a `std::unique_ptr`, but it can be moved.

```
T& getbin(T &obj)           // copying to existing variable
```

The second version takes a reference to an existing object and copies the binary data directly into it. Use this version to get stack allocation of binary data. Both versions of `getbin` takes an optional `size_t&` to get the actual size of the binary object.

```
sa::tuple res = ...;
char buf[128];
a_struct as;
res[0].getbin(buf);                     // copy binary object to buf
res[1].getbin(as);                      // copy binary object to as

auto p = res[0].getbin<uint64_t>();     // dynamic allocation with new
if (*p == 0) ...                        // p behaves like uint64_t*

auto b = res[1].getbin<a_struct>();     // dynamic allocation with new
if (b->mem == 0) ...                    // b behaves like a_struct*

auto a = res[2].getbin<char[16]>();     // dynamic allocation with new
if (a[1] == 'x') ...                    // a behaves like char[]

} // scope exit: p, b and a are deleted automatically
```

To set an object to a binary value, use either of

```
setbin(bin<T>)              // setting from a binary blob

setbin(const &T)            // setting from existing variable
```

Class sa::tuple

A tuple is, much like the `std::tuple`, a heterogenous collection of objects and is itself an `object`. A `tuple` may be created from primitive types or objects, for example

```
extern double pi;
sa::tuple tpl{ 1, "a1", a_real, pi };
int width = tpl.size();     // returns 4
```

The individual elements of a tuple can be accessed using the [] operator, like

```
tpl[3] = true;
sa::object elem1 = tpl[0];
std::string str{ tpl[1].gets() };
```

Since a tuple is not a native container, the [] operator does not return a reference to an object, but rather an iterator (tuple::itor), which is conceptually an object. The iterator can also be used in for loops:

```
for(auto o : tpl) o.print();
```

This is useful if the tuple happens to represent a homogenous array of numbers, a vector.

Class sa::record

A record is a key-value mapping, where the key is a string. The interface is similar to the tuple interface, except that the [] operator takes a string (const char*):

```
sa::record rec{ {“key1”, a_real}, {“key2”, tpl} };
rec[“key3”] = “the string”;      // add new pair to record
double k = rec[“key1”].getd();   // get the value indexed by “key1”
```

Class sa::connection

A connection is used to send commands and queries to sa.engine. Before a connection can be created, an *embedded* sa.engine must be initialized using

```
sa::engine_init()   // default parameters gives default settings
```

After this call, a connection can be created with either of (for details, see the sa_CAPI 2.0 documentation):

```
sa::connection c(“”);            // embedded connection
sa::connection c(“peer”);        // local connection
sa::connection c(“peer@host”);   // remote connection
```

Once the connection is created, queries and commands can be sent:

```
sa::tuple res = s.runquery(“sqrt(3)”);
res = s.runcall(“sqrt”, sa::tuple{3.0});
sa::stream s = c.query(“range(1,3)”);
```

Class sa::stream

The result of a connection query or call is a sa::stream. The stream can either be run() to get the return value directly in a tuple, or mapped to a callback function for streaming data. For example

```
sa::tuple res = s.run();  // returns a tuple {3} for “range(1,3)”

int sum = 0;
s.map([&sum](const tuple_view &t) {     // lambda to compute a sum
  sum += t[0].geti(); return true;
});
// sum will be 6 for the query “range(1,3)”
```

The callback function should return true to keep the data coming, or false to stop the stream. The parameter type tuple_view is a variant on tuple that does not take ownership of the tuple. It is used for callback parameters.

There are several flavors of the map callback. The one showed in the previous example is a capturing lambda expression. Of course, any function or functor matching the signature bool(const tuple_view) will do, not only lambda expressions. There is also a templated type-safe non-capturing version that would look like this:

```
int sum = 0;
```

```
s.map<int>(sum, [](const tuple_view &t, int &s) {
  s += t[0].geti(); return true;
});
```

Notice that the types of the first parameter to `map` and the second parameter of the callback have the same base type as the `map` template type. Non-capturing lambdas has the advantage that they are convertible to normal function pointers.

There is also the possibility to use a static C function for the callback, like this:

```
ohandle cb(sa_tuple tpl, void *x) {
  sa::tuple_view t{tpl};  // remember to use tuple_view for callbacks
  int *sum = (int*)x;
  *sum += t[0].geti();    // may throw if not an integer
  return true;            // return true to continue the stream
}
...
int sum = 0;
s.map(&sum, cb);
```

Notice that the first parameter of the callback is a raw C `ohandle` in disguise that must not be freed. This is the purpose of using tuple_view to get the tuple contents. Using static C functions as callbacks relies on C-style casts and is not type safe and is therefore not recommended for general use.

To run `stream::map()` with a class member function, use a static C function or a lambda expression to redirect the callbacks to the class like this:

```
class A {
  void do_it(int i) { ... }
} a;

static int a_cb(sa_tuple tpl, void *x) {
  A* a = (A*)x;
  sa::tuple_view t{tpl};
  a->do_it(t[0].geti());
  return TRUE;
}
// ...
c.map(&a, a_cb);    // map with static C callback

// ... or map with non-capturing type safe lambda:
c.map<A>(a, [](const sa::tuple_view t, A &a) {
  a.do_it(t[0].geti()); return true;
});

// ... or map with capturing lambda:
c.map([&a](sa::tuple_view t) { a.do_it(t[0].geti()); return true; });
```

Class sa::callcontext

Foreign functions in C++ are declared as

```
ohandle my_function(sa::callcontext cc)
```

The call context cc can be used to extract arguments, set unbound variables, and to emit data to the system. For example

```
ohandle my_function(sa::callcontext cc) {
  try {
    double arg1 = cc[1].getd();  // get first argument
    int arg2 = cc[2].geti();     // get second argument
    cc[3] = pow(arg1, arg2);     // bind a value to unbound arg
    cc.emit();                   // emit the return value
  }
  catch (sa::error&) {}
  return nil;
}
```

Observe that the indexing of arguments and unbound return values starts at 1. If an argument is not available, or it is not of the expected type, or element [0] is accessed, an `sa::error` exception will be thrown.

Call contexts receiving a stream can use a map function much like sa::stream does, and has the same syntax except for the function signature, which for a C function is

```
ohandle callback(a_callcontext cxt, int width, ohandle o[], void *xa);
```

Using lambda expressions, use for example

```
double sum = 0.0;
cc[1].map([&sum](int len, const object_view o[]) {
  if (len > 0) { double x = o[0].getd(); sum += x*x; }
});
```

That’s all, folks!

Interface implementation for things semantically an object

	object	tuple::itor	record::itor	callctx::itor
h()	Yes[2]	Yes	Yes	Yes
isvalid	Yes[2]	Yes	Yes	Yes
operator!	Yes[2]	Yes	Yes	Yes
type_t type()	Yes[2]	Yes	Yes	Yes
isint	Yes[2]	Yes	Yes	Yes
isdouble	Yes[2]	Yes	Yes	Yes
isstring	Yes[2]	Yes	Yes	Yes
isbool	Yes[2]	Yes	Yes	Yes
isbinary	Yes[2]	Yes	Yes	Yes
isarray	Yes[2]	Yes	Yes	Yes
istuple	Yes[2]	Yes	Yes	Yes
isrecord	Yes[2]	Yes	Yes	Yes
issurrogate	Yes[2]	Yes	Yes	Yes
= (int)	Yes[1]	Yes	Yes[4]	Yes[4]
= (int64_t)	Yes[1]	Yes	Yes[4]	Yes[4]
= (float)	Yes[1]	Yes	Yes[4]	Yes[4]
= (double)	Yes[1]	Yes	Yes[4]	Yes[4]
= (const char*)	Yes[1]	Yes	Yes[4]	Yes[4]
= (const string&)	Yes[1]	Yes	Yes[4]	Yes[4]
= bool	Yes[1]	Yes	Yes[4]	Yes[4]
= object	Yes[3]	Yes	Yes	Yes (const&)
= record	Yes[3]	Yes	Yes	Yes (const&)
= tuple	Yes[3]	Yes	Yes	Yes (const&)
operator object()		Yes	Yes	Yes
operator*		Yes		
geti	Yes	Yes	Yes	Yes
getl	Yes	Yes	Yes	Yes
getf	Yes	Yes	Yes	Yes
getd	Yes	Yes	Yes	Yes
gets	Yes	Yes	Yes	Yes
getb	Yes	Yes	Yes	Yes
geth				
geto				
geta	Yes	Yes	Yes	Yes
getr	Yes	Yes	Yes	Yes
getbinsize	Yes	Yes	Yes	Yes
op = bin<T>	Yes[1]	Yes	Yes	Yes
bin<T> getbin	Yes	Yes	Yes	Yes
getbin(T&)	Yes	Yes	Yes	Yes
setbin(bin<T>)	Yes	Yes	Yes	Yes
setbin(const &T)	Yes	Yes	Yes	Yes
print	Yes[2]	Yes	Yes	Yes
error	Yes[2]	Yes	Yes	Yes

[1] implemented through ctor

[2] implemented in base class handle

[3] implemented through taking handle

[4] implemented through taking object

Oddities (code auto-review):

- Both sa_client.h and sa_core.h are always included. This clutters the default namespace.
- sa::error does not inherit std::exception or any of its descendants. Why should it?
- Why are handle and object two different classes?
- Is object really ever needed? So far, I found only to put a binary in a tuple, and that is more of a lacking feature in the tuple ctor. It is also used as an intermediate to avoid code in the implementation of especially record.
- Here and there I use type from stdint. The problem with that is that it does not map well for templates. For example, in MSVS, 42LL is an int64_t, but not in gcc. So int64_t cannot be used for template matching, but is the return value of several functions.
- bin<T> is basically a std::unique_ptr<T>. But std::unique_ptr<T> initialization doesn't work in C++11, and there is no way of handling arrays with a fixed element count like char[128] even in later versions of C++.
- bin<T> could be made smart enough to automatically allocate on the stack when the object is small enough.
- The handle class has many friends. I think it is better with explicit friends than to make _h accessible for users. The current interface is safe even if the user inherits from one of the classes.
- The object class does not have assignment operators, but relies on the constructors.
- The binary setters/getters in object and tuple rely on the thread safe interface in sa_client.h, as there is no viable alternative in sa_core.h.
- An object_view is rather non-const for a _view class, but I don't know of any efficient way of disabling parts of the interface. Also, it might be ok to modify the view object.
- A record cannot be indexed with a std::string, because the strings lifespan might not exceed the iterators, which means the c_str() could be invalid when used.
- The tuple::itor and callcontext::itor interfaces should ideally be identical to the object interface except for some extra operations like ++. This could be enforced with an abstract interface base class or a CRTP, which has not been implemented.
- tuple has getarity() and size(), while callcontext has arity().
- There are many map interfaces; plain C function, template version for type safety (non-capuring only), or std::function for capturing lambdas. Which should be available?
- I hooked the error reporting functions onto sa::handle and callcontext::itor. Since they need a handle parameter anyway, this seems like the logical spots.
- connection has overloads to deal with r-value references, but not the other classes. Apparently, automatic cast to l-value reference is a MSVS extension, and is not available in gcc by default. Should this be fixed?
- connection::query() does not take a std::string.
- The details section could be extracted to a separate header to hide it a bit.
- Not all functions have thread-safe/unsafe versions.
- The getstring C API functions are inconsistent in that the thread safe variants return a string including the zero termination, and thus have greater length than expected. There is a fix for that in the sa_interface, but should perhaps be fixed in sa_client.h instead.
- There is a difference in core and client versions of handle, in that the core version takes an extra refcount on the handle. This is because the object creation functions work differently, and this was the most convenient place to remedy that. However, should a user start creating her own handles, then there could be leak problems.

- object(bool) is inconsistent in that the !thread_safe version uses falsesymbol for false, and most of the other functions uses nil.