A NEMETSCHEK COMPANY

3D FOR THE
REAL WORLD.

DevKitchen 2018
MAXON API - Basic Tools

05.11.18

MAXON API
Basic Tools

- Error Handling
- Configuration Variables
- Data and DataDictionary
- References
- Delegates
- Observables and Observers
- Threading
- Logger
- Q&A

Error Handling in MAXON API

Designed for:

- Simplicity and flexibility
- Code robustness and readability
- Efficiency
- Thread safety
- Enforceable by compiler

Based on `maxon::Result<T>` and `maxon::Error`

Features:

- Unified system for returning errors and values in parallel
- Optimized for speed, e.g. errors can even be preallocated
- Errors may be aggregated
- Errors may contain additional information, like:
  - Error type (nullptr error, illegal argument, …)
  - Source location
  - Error string

Common Practice in Error Handling

- A method which might fail should return a `maxon::Result<T>`
- A caller should (actually has to) handle the error
- Good practice, special variants of `maxon::Result<T>`
  - Even check the result if a method can not fail (it may later…)
    - see `maxon::ResultOk<T>`
  - Explicit handling of out-of-memory situations
    - see `maxon::ResultMem<T>`, `maxon::ResultRef<T>`,…

Returning Errors, use of `maxon::Result<T>`

```
static maxon::Result<maxon::Int> GetRandomNumber()
{
  maxon::Int number = 0;

  if (RollDice(number) == false)
    return maxon::UnknownError(MAXON_SOURCE_LOCATION, "Could not roll the dice."_s);

  return number;
}
```

- Without return value use return type `maxon::Result<void>`
  - `maxon::OK` is then returned in case of no error

Handling Errors the Simple Way: `iferr_return`

```
static maxon::Result<maxon::Int> GetRandomNumberTimesTen()
{
  // declare error scope
  iferr_scope;

  // return if an error is returned, forwarding the error automatically
  const maxon::Int randomNumber = GetRandomNumber() iferr_return;

  return randomNumber * 10;
}
```

- Don’t forget to call `iferr_scope` for `iferr_return` to work

Cleanup After Errors: `iferr_scope_handler`

```
static maxon::Int GetRandomNumberLegacy()
{
  iferr_scope_handler // is called by iferr_return and iferr_throw
  {
    DiagnosticOutput("GetRandomNumberLegacy() error: @", err);
    return 0; // return value in error case
  };
  const maxon::Int randomNumber = GetRandomNumber() iferr_return;
  if (other error)
    iferr_throw(maxon::UnknownError(MAXON_SOURCE_LOCATION, “Really?“_s))
  return randomNumber;
}
```

- Nice to convert errors to legacy code, also note `finally`

Executing Code After Errors: `iferr` and `ifnoerr`

```
static maxon::Result<maxon::Int> GetRandomNumberTimesHundred()
{
  maxon::Int randomNumber = 0;

  // assign the return value while evaluating the maxon::Result
  iferr (randomNumber = GetRandomNumber())
  {
    return err; // "err" is defined in iferr
  }
  return randomNumber * 100;
}
```

More Error Handling Than We Can Cover in Ten Minutes

- Macros to explicitly mark results not checked: `iferr_cannot_fail()` and `iferr_ignore()`
- Additional to `iferr_scope_handler` there is also `finally`
- Utility macros: `CheckArgument()`, `CheckState()` and `CheckAssert()`
- Integrates with `JobInterface` or threads, result: aggregated error

- Take Away: Check for Errors! Always! No excuses!

Error Handling in SDK Documentation

- Error System Overview
  - Error Handling Manual
  - Error Result Manual
  - Error Class Manual
  - Error Utility Manual
  - Error Types Manual

Configuration Variables in MAXON API

- Unified system to introduce configuration variables into Cinema 4D
- A configuration variable can be set in different ways:
  - as environment variable
  - via command line
  - via `config.txt`
- Supported data types: Bool, Int, Float and String
- Support default values and can be tested for existence
- Support for help strings

Configuration variables - A Quick Example

```
MAXON_CONFIGURATION_STRING(g_myStringVar, "", maxon::CONFIGURATION_CATEGORY::REGULAR,
                           "Set a string");
MAXON_CONFIGURATION_BOOL(g_myBoolVar, false, maxon::CONFIGURATION_CATEGORY::REGULAR,
                         "Set to true or false.“);
// your configuration variables may accessed directly
if (g_myStringVariable.IsEmpty())
  return maxon::OK;

// or an arbitrary configuration variable can be queried
maxon::CONFIGURATIONENTRY_ORIGIN origin;
maxon::CONFIGURATIONENTRY_STATE  state;
maxon::Bool boolVar = false;
if (maxon::Configuration::QueryBool("g_myBoolVar"_s, boolVar, origin, state))
  // ...
```

Configuration Variables in SDK Documentation

- Configuration Variables Manual

`Data` in MAXON API

- Similar to `GeData`, just better
- Can hold any MAXON Data Type
- Supports error handling
- Generic getting and setting of data
- Better comparison and conversion support
- Support for hash values
- `ToString()` nice for e.g. debugging

`DataDictionary` in MAXON API

- A more powerful `BaseContainer`
- Based on `maxon::DataDictionaryInterface`
- Stores arbitrary MAXON API Data, referenced by keys
- Supports error handling
- Generic getting and setting of data
- Iterateable
- `ToString()` nice for e.g. debugging

Using Data and DataDictionary is Straightforward

```
maxon::DataDictionary dataDict;

// simple set
dataDict.Set(0, maxon::Int32(100)) iferr_return;
// set maxon::Data
const maxon::Data number(maxon::Int32(123));
dataDict.SetData(maxon::ConstDataPtr(1), std::move(number)) iferr_return;

// simple get
const maxon::Int32 value = dataDict.Get<maxon::Int32>(0) iferr_return;
DiagnosticOutput("Value: @", value);
// get maxon::Data
const maxon::Data data = dataDict.GetData(maxon::ConstDataPtr(1)) iferr_return;
DiagnosticOutput("Data: @", data);
```

Iterating a `DataDictionary` - As Expected

```
for (const auto& data : dataDict)
{
  const maxon::Data key = data.first.GetCopy() iferr_return;
  const maxon::Data value = data.second.GetCopy() iferr_return;
  const maxon::DataType* const type = value.GetType();

  DiagnosticOutput("Key: @, Value: @ (@)", key, value, type);
}
```

`Data` and `DataDictionary` in SDK Documentation

- MAXON Data Type Manual
- Data Manual
- DataDictionary Manual
- MAXON API Containers & Data Collections Overview
  - BaseArray Manual
  - HashMap Manual
  - BaseList Manual
  - BaseSort Manual

References in MAXON API

- Similar to smart pointers in C++
- Allow to express ownership
- Allow to share objects, reference counted
- Take care of object destruction

- Benefits: Safer code, less prone to memory leaks, less book keeping

Types of References in MAXON API

- `UniqueRef` Similar `std::unique_ptr`
  Classic API: `AutoObj()` or `AutoMem()`
- `StrongRef` Shared reference, similar `std::shared_ptr`, reference counted, actually NIRC…
- `StrongCOWRef` Like `StrongRef`, just COW
- `WeakRef` Similar `std::weak_ptr`
  Can return `StrongRef`, if object still exists

General Rules for References

- Potentially dangerous conversions are prevented

• `UniqueRef`	Can be moved, but not copied
• `StrongRef`	Can not be assigned to a `UniqueRef` Can be moved or copied
• `StrongCOWRef`	Same as for `StrongRef`
• `WeakRef`	Can reference objects referenced by `UniqueRef` or `StrongRef`

Consistent Object Creation, Using `StrongRef`

```
class TestClassWithConstructor
{
  TestClassWithConstructor(Int) { ... }
};


using TestRef = maxon::StrongRef<TestClassWithConstructor>;

TestRef t1 = TestRef::Create(42) iferr_return;

TestRef t2;
t2.Create(42) iferr_return;
```

NIRC - Non-Intrusive Reference Counting

- References and Cinema 4D's core memory management care for reference counting, objects don't need to implement any code
- Weak references automatically supported
- Benefits: Simplified threading, less code
- Possibility to implement custom reference counting
  - for e.g. alien memory management, debugging, etc.

Properties of `WeakRef`

- Does NOT prevent destruction of referenced object
- Can be used to point to objects not owned
- Can create a `StrongRef` with very little overhead
- Possibility to add Observers (functions called on certain events)
  - invoked before object's destruction
  - may claim ownership (e.g. for later re-use)
- Can be used to track lifetime of an object

Checking Lifetime of a Raw Pointer via `WeakRef`

```
static Bool WasAlive(maxon::WeakRefBase& weak, const void* object)
{
  const void* target = weak.data->Lock();
  Bool wasAlive = target == object;
  weak.data->Unlock(target);
  return wasAlive;
}
```

References in SDK Documentation

- References Manual

Delegates in MAXON API…

- replace C-style callbacks
- are similar to `std::function`, but with error handling
- avoid dangerous pointer casts
- are binary stable
- can encapsulate C function pointers, C++ pointers to member functions, lambdas and any MAXON API interface method
- are available in Cinema 4D R19, already

Using a Delegate, Actually as Simple as That

```
// function using a Delegate to provide progress information
static void LongTask(const maxon::Delegate<maxon::Bool(maxon::Int32)>& progress)
{
  for (maxon::Int32 i = 0; i < 100; ++i)
  {
    const maxon::Bool res = progress(i); // call the delegate

    if (!res)
      return;
  }
}
```

Multiple Ways to Define Delegates

```
auto lambda = [](maxon::Int32 p) // use a lambda object
              {
                DiagnosticOutput("Percentage: @", p);
                if (p > 66)
                  return false;
                return true;
              };
LongTask(lambda); // of course the lambda could also be passed directly

// use maxon::Delegate object
const maxon::Delegate<maxon::Bool(maxon::Int32)> maxonDelegate(SomeFunction);
LongTask(maxonDelegate);

LongTask(SomeFunction); // or just a function pointer
```

Delegates in MAXON API - Calling Conventions

- Constructors must use `&&` for `Delegate` parameters
- Methods should use `&&` for `Delegate` arguments, exception: Methods executing a `Delegate` directly
- Return `Result<T>` if a method copies a `Delegate`
- Return a `Delegate` via `std::move`
- „Getters“ should return `const Delegate<>&`
- Lambdas may capture a `Delegate` by reference, by move capture or via `CopyWrapper<>`

Delegates in MAXON API - A Word of Warning

- Keep referenced objects alive until the `Delegate` will be destructed
  - when a `Delegate` encapsulates a pointer-to-member-function
  - when a `Delegate` is constructed via `CreateByReference()`
  - when an encapsulated lambda has capture objects by reference

- Failing to do so will result in SDK Team denying any support.
  - Just kidding, of course.
  - But these errors are really hard to debug!

Delegates in SDK Documentation

- Delegates Manual

Observables and Observers in MAXON API

- An Observable is a registry storing references to multiple functions, the Observers, which are called in case of certain events

- Will probably replace `EVMSG_CHANGE` in the long run

- Based on `ObserverObjectInterface`
- Interfaces can declare multiple Observables

Declare an Interface with an Observable

```
class ObserveMeInterface : MAXON_INTERFACE_BASES(maxon::ObserverObjectInterface)
{
  MAXON_INTERFACE(ObserveMeInterface, MAXON_REFERENCE_NORMAL,
                  "net.maxonexample.interfaces.observeme");

public:
  // Arbitrary function makes creating an event to be observed
  MAXON_METHOD void OnClick();

  // The Observable, functions registered here will be called, when Ping() is called
  MAXON_OBSERVABLE(void, ObservableOnClick, (maxon::Int32 count),
                   maxon::ObservableCombinerRunAllComponent);
};
```

Implement Observable Functionality I

```
class ObserveMeImpl : public maxon::Component<ObserveMeImpl, ObserveMeInterface>
{
  MAXON_COMPONENT(NORMAL, maxon::ObserverObjectClass); // implements observer funct.
private:
  maxon::Int32 _count;
public:
  MAXON_OBSERVABLE_IMPL(ObservablePing); // implements the observable
  maxon::Result<void> InitComponent()
  {
    iferr_scope;
    _count = 0;
    _ObservableOnClick.Init(self, maxon::Id("ObservablePing")) iferr_return;
    return maxon::OK;
  }
```

Implement Observable Functionality II

```
  void FreeComponent()
  {
    _ObservablePing.Free(); // free observable
  }

  MAXON_METHOD void OnClick()
  {
    ++_count;
    iferr (_ObservablePing.Notify(_count)) // notify subscribers aka Observers
    {
      DiagnosticOutput("Observable error: @", err);
    }
  }
};
```

Registering Observers

```
// The Observer, has to match Observable in return value and parameters
static void ReactOnClick(maxon::Int count)
{
  DiagnosticOutput("Count: @", count);
}


// Add the observer to the "ObservablePing" observable
// This can be done in multiple ways, as already shown for Delegates
g_observeMe.ObservablePing().AddObserver(ReactOnClick) iferr_return;


// Calling the OnClick() function will notify all registered observers
g_observeMe.OnClick();
```

Observables in SDK Documentation

- Observables Overview
  - Observables Declaration Manual
  - Observables Implementation Manual
  - Observables Usage Manual

Threading in MAXON API

- Multiple options: `ParallelFor`, Jobs, Threads
- Actually not new, in use in Cinebench R15 for a while
- Scales excellently

- Synchronization, the usual suspects
  - Locks, `Synchronized` and `Serializer`
  - Condition Variables
- Or even better: References and good design

Parallelization the Simple Way: `ParallelFor`

```
maxon::BaseArray<maxon::Int32> results; // something to store thread results
results.Resize(count) iferr_return;

auto worker = [&results](maxon::Int i) // our „thread code“
              {
                // ...  do work per index i here ...
                results[i] = 42; // with above work, the best result ever
              };

// kick-off the threaded execution
maxon::ParallelFor::Dynamic(0, count, worker); // from, to, lambda [,  ...]

for (const maxon::Int32 res : results) // evaluate the results
  DiagnosticOutput(": @", res);
```

`ParallelFor` - Simple, Not Weak

- Variants: `Dynamic` and `Static`
  - Dynamic to be preferred, usually better performance

- Additional lambdas and modes for initialization and finalization
- Additional parameters: thread count, granularity, job queue

- also `ParallelImage` and `ParallelInvoke`

Jobs - Threaded, Reusable, Independent Work Items

- Based on `JobInterfaceTemplate`, `JobResultInterface` and `JobInterface`
- Support Observers (called when job has finished)
- May be organized in job groups
- May be assigned to different queues (usually not necessary)

- Jobs **must not** have additional, hidden dependencies!
  Like e.g. internal parallelized code

Creating a Job

```
class MyJob : public maxon::JobInterfaceTemplate<MyJob, maxon::UInt>
{
public:
  MyJob() { };
  MAXON_IMPLICIT MyJob(maxon::UInt n)
  { _n = n; }

  maxon::Result<void> operator ()() // function operator with the workload
  {
    // ... do some work...
    return SetResult(std::move(result)); // store result
  }
private:
  maxon::UInt _n = 0;
};
```

Enqueueing Jobs and Evaluating Result, Cancelling Jobs

```
// create and start job
const maxon::UInt n = 42;
auto job = MyJob::Create(n) iferr_return;
job.Enqueue();  // enqueue the job in the current queue and start

// wait and get result
const maxon::UInt result = job.GetResult() iferr_return;
DiagnosticOutput("@", result);



// or cancel the given job
job.CancelAndWait();
```

Using a Group of Jobs

```
// create group
maxon::JobGroupRef group = maxon::JobGroupRef::Create() iferr_return;

for (maxon::UInt n : numbers)
{
  // create and add job
  auto job = MyJob::Create(n) iferr_return;
  group.Add(job) iferr_return;
}

// enqueue jobs and wait
group.Enqueue();
group.Wait();
```

Custom Threads - Of Course Possible, too

- Based on `ThreadInterfaceTemplate`, `ThreadInterface` and `JobInterface`

Creating a Custom Thread

```
class MyThread : public maxon::ThreadInterfaceTemplate<MyThread>
{
public:
  const maxon::Char* GetName() const { return "MyThread"; }

  maxon::Result<void> operator ()()
  {
    // ... your code goes here ...
    // ... don't forget to check IsCancelled()
    return maxon::OK; // or an error
  }
  maxon::Result<whatever> GetResult() // good practice
  {
  }
};
```

Starting a Custom Thread

```
// Usually a global static variable to store the custom thread.
static maxon::ThreadRefTemplate<MyThread> g_myThread;

if (!g_myThread) // create and start thread
{
  g_myThread = MyThread::Create() iferr_return;
  g_myThread.Start() iferr_return;
}
else
{
  g_myThread->CancelAndWait(); // cancel thread if still running
  const whatever result = g_myThread->GetResult() iferr_return; // get result
  g_myThread = nullptr; // clear thread
  // ... use result...
}
```

Jobs and Threads - A few Words of Warning

- Jobs and Threads are reference counted

Should actually go without saying:

- **Must not** be created directly on stack
- **Must not** be created as member variables of a class

- Longer running jobs and threads should periodically call `IsCancelled()`

Threading in SDK Documentation

- MAXON API Threading Overview
  - Parallel Manual
  - Jobs Manual
  - Threads Manual
  - Locks & Synchronization Manual
  - Condition Variables Manual

Logger in MAXON API

- Based on `LoggerInterface`
- Output is now directed to different Loggers
- These Logger appear as different categories in the new Console
- Different Logger „audiences“ are available, e.g. users or debugger
- Logger messages support certain meta data, e.g. severity of errors
- Multiple built-in loggers are provided, e.g. for Python
- Logger support Observers
- Custom Loggers are possible, e.g. for a plugin

Creating a Custom Logger

```
static maxon::LoggerRef g_exampleLogger; // IMPORTANT: set to nullptr when Cinema shuts down
const maxon::Id loggerID("net.maxonexample.logger"); // define logger ID
if (!maxon::Loggers::Get(loggerID)) // check if the logger already exists
{
  iferr (g_exampleLogger = maxon::LoggerRef::Create()) // create the new logger
  {
    g_exampleLogger = maxon::Loggers::Default(); // on error use default logger
  }
  else
  { // add the logger type "Application" to display logged messages in the "Console"
    g_exampleLogger.AddLoggerType(maxon::TARGETAUDIENCE::ALL,
                                  maxon::LoggerTypes::Application()) iferr_return;
    g_exampleLogger.SetName("Example Logger"_s);
    maxon::Loggers::Insert(loggerID, g_exampleLogger) iferr_return; // insert in registry
    const maxon::Int32 BFM_REBUILDCONSOLETREE = 334295845;
    SpecialEventAdd(BFM_REBUILDCONSOLETREE); // IMPORTANT: update Console window
  }
}
```

Clean-up for a Custom Logger

```
static void FreeExampleLogger()
{
  g_exampleLogger = nullptr;  // reference must be freed

  // make sure to also remove the entry of your logger from Loggers.
  maxon::Loggers::Erase(maxon::Id("net.maxonexample.logger")) iferr_ignore("Soooo?");
}

MAXON_INITIALIZATION(nullptr, FreeExampleLogger);
```

Using a Logger

```
// write a message to the default logger.
ApplicationOutput("Some Message."_s);

// write to a custom logger referenced in g_exampleLogger
g_exampleLogger.Write(maxon::TARGETAUDIENCE::ALL, "Foo"_s, MAXON_SOURCE_LOCATION,
                      maxon::WRITEMETA::CRITICAL) iferr_return;

// or write to the logger by getting the logger using the logger ID
// get logger by ID
const maxon::Id         loggerId("net.maxonexample.logger");
const maxon::LoggerRef logger = maxon::Loggers::Get(loggerId);

logger.Write(maxon::TARGETAUDIENCE::ALL, "Bar"_s, MAXON_SOURCE_LOCATION,
             maxon::WRITEMETA::DEFAULT) iferr_return;
```

Logger in SDK Documentation

- LoggerInterface Manual

Thanks for Your Interest and Patience!

Any Questions?

3D FOR THE REAL WORLD.

MAXON Computer GmbH
Max-Planck-Straße 20
D-61381 Friedrichsdorf
Germany

E info@maxon.de
T +49 (0) 6172 5906-0
F +49 (0) 6172 5906-30

maxon.net

05.11.18