++Skype library
Tutorial

St. Petersburg
2006

Contents

1 Introduction

The ++Skype is a C++ library of thoroughly designed classes. It can help you in several situations. Do you need to develop a platform independent Skype add-on software? Try ++Skype! Do you want to become an expert in the low-level Skype API implementations (D-BUS, Windows messages)? If no, try ++Skype!

Key library features:

- Platform independence (Linux and Windows are supported this time);
- Easy to use;
- Easy to extend because of a flexible library design inspired by modern C++ design ideas;
- Performance was one of our goals — only compile-time polymorphism is used;
- Open source, licensed under LGPL v. 2.1

The aim of this tutorial is to help you familiarize with the library. The ++Skype installation process is not covered by this document, you can find installation instructions here.

2 ++Skype architecture

There are four parts of the library — *low-level components, high-level components, utilities* and *exceptions.*

Low-level layer is the heart of the ++Skype. High-level components are based on the low-level components. Howerever, it is possible to use low-level components wholly on their own, without high-level components. Sometimes this feature is very helpful.

Low-level components are:

RawSkypeConnection (Trivial) connection to Skype.

SkypeConnection Connection to Skype with support of error handling and callback functors.

High-level components use *SkypeConnection* to interact with Skype. These components represent various Skype objects, such as calls, chat, chat messages, privileges, applications etc.

High-level components are:

SkypeObject and SkypeObjManager These classes are the basis for all other high-level components. All core functionality is implemented in these classes.

Call and CallManager Abstractions of Call and manager of calls.

Chat and ChatManager Abstractions of Chat and manager of chats.

ChatMsg and ChatMsgManager Abstractions of Chat message and manager of chat messages.

AppStream and Application Abstractions of Application and application data stream.

User and UserManager Abstractions of User and manager of users.

AudioSettings This class encapsulates Skype audio settings such as audio devices, echo cancellation policy and so on.

Privileges This class encapsulates current Skype user privileges (SkypeIn, SkypeOut, VoiceMail).

Profile This class encapsulates current Skype user profile.

All high-level components are put into the namespace *Ice*.

In order to inform the application on critical errors, exceptions are used. All ++Skype exceptions are declared in the `SkypeExceptions.h` header. The exception hierarchy is simple enough.

Utilities are used by all the ++Skype library layers. This is the last but not least part of the library.

The utilities are:

Loggers Different logging classes — file logging, syslog logging[1] and loggers chain are supported.

Runtime assertions C++ wrapper for well-known C-function `assert`.

String to numeric types conversion template Converts the string which contains the number into numeric type.

We will start our discussion from the utilities because they are used by all the ++Skype classes. After this, we will look at the exception hierarchy, low-level components and high-level components.

2.1 Utilities

2.1.1 Loggers

Logging is a very important aspect for any software project. Sometimes, log files are the only source of information related to the software crash or critical errors. ++Skype logging facilities are very flexible. Each ++Skype class which is able to log inherits from `LogAbility` class. This class is declared in `Loggers.h` library header:

```
typedef
enum { LlDbg = 0, LlInfo, LlWarning, LlError, LlCritical } LogLevel_t;

typedef Loki::Functor<void,
LOKI_TYPELIST_3(const LogLevel_t, const char*, va_list)> LogFunc_t;

class LogAbility {
public:
        LogAbility(const LogFunc_t& _logger = DummyLogger);
        void log(LogLevel_t _lvl , const char* _fmt, ...) const;
        const LogFunc_t& GetLog() const;
};
```

`LogLevel_t` is the level of log-message. `LogFunc_t` is functor [1, ch. 5] returning void with three arguments of types `LogLevel_t, const char*` and `va_list`. `LogAbility` constructor gets the argument of type `LogFunc_t`. Default value of this argument is `DummyLogger`, which does no logging at all. The `log` member function of the `LogAbility` class calls functor of type `LogFunc_t` (supplied in the constructor of the `LogAbility`) to log message. The signature and format convention for `log` member function is exactly the same as for the well-known `printf` function.

You'll tell me: "Oh, no! It looks too difficult. I should always define my own logger functor to use the library!". Don't worry. You don't have to do this yourself. Loggers are already implemented in the ++Skype. But if you want to implement your own logger, it is no problem to use it with any ++Skype class. This is exactly what the word "flexibility" means. The following loggers are implemented in the library: `FileLogger, StreamLogger` and `LoggersChain`. Logging through syslog is available in the linux distribution of the ++Skype. The corresponding logger is called `SysLogger`. And of course, don't forget about the `DummyLogger!`

`FileLogger, StreamLogger` and `SysLogger` are templates with the only template parameter `Prefix`, "prefix policy". What is the meaning of this parameter? Every log-message is prefixed. The prefix contains a lot of helpful information related to the message such as log level, date and time etc. The following public operator should be defined in the implementation of the prefix policy:

[1] Only in linux version of the ++Skype.

```
std::string& operator()(const LogLevel_t& _lvl);
```

This operator is called by the logger to produce prefix for the message. There are three prefix policies implemented in ++Skype — `NoLogPrefix`, `LvlLogPrefix` and `LvlTimeLogPrefix`. You can implement your own prefix policy. It's easy! The default prefix policy is `NoLogPrefix`.

`FileLogger`, `StreamLogger`, `SysLogger` and `LoggersChain` have the following member function:

```
void Do(const LogLevel_t _lvl, const char* _fmt, va_list _args);
```

The signature of this member function is the same as required by `LogFunc_t` definition. That's why you can use these loggers as ++Skype logger functors. Look at the following `StreamLogger` usage example:

```
typedef StreamLogger<LvlTimeLogPrefix> MyLogger_t;
MyLogger_t logger(std::cout);

LogFunc_t lf(&logger, &MyLogger_t::Do);
```

`FileLogger`, `StreamLogger` and `SysLogger` inherit from the `BaseLogger` template. The most useful feature of this template is its `SetMinLogLvl` public member function. Use it to implement "distributed logging" – different loggers log messages with different severity.

The last class we've not discussed yet is `LoggersChain`. This class is used to represent the chain of loggers. You can add loggers or functors of type `LogFunc_t` to the chain and remove them from the chain. `Do`–function of the chain executes all the log-functors stored in the chain. `LoggersChain` is declared in `Loggers.h` library header:

```
class LoggersChain : public LogAbility {
public:
        LoggersChain() throw();
        template <typename L> void AddLogger(L& _logger);
        void AddLogFunc(const LogFunc_t& _logger);
        void PopBackLogger(void);
        void Do(const LogLevel_t _lvl, const char* _fmt, va_list _args);
};
```

To add a ++Skype logger into the chain, call `AddLogger` member function. To add a logger functor of type `LogFunc_t`, call `AddLogFunc` member function. To remove the last logger from the chain, call `PopBackLogger` member function. To represent the chain as functor of type `LogFunc_t`, use pointer to the `Do`–function, as shown below:

```
LoggersChain loggers;

LogFunc_t lf(&loggers, &LoggersChain::Do);
```

It is a good design practice to declare `LoggersChain` as a singleton [1, ch. 6] and to store references to the loggers used by the application inside this object. Note, `LoggersChain` doesn't store the loggers itself, it stores only functors!

2.1.2 Runtime assertions

`Assert` macro is defined in `utils.h` library header:

```
#define Assert(tst) AssertFunc<DbgMode>(tst,
        "AssertionViolation was thrown at " FILELINE);
```

There are two modes of ++Skype compilation and usage – *debug* and *release*. To switch between these modes you have to define/undefine the `NDEBUG` preprocessor variable. If `NDEBUG` is defined, ++Skype mode is 'release', otherwise it is 'debug'.

`AssertFunc` tests the logical expression `tst` *if and only if* ++Skype is in debug mode. If `tst` is true or ++Skype is in the release mode, nothing happens. If `tst` is false and ++Skype is in the debug mode, `AssertionViolation` exception is thrown.

2.1.3 String to numeric types conversion template

This template is declared in `utils.h` library header:

template <**class** C> C GetFromString(**const** std::string& value);

The usage is obvious. Just call `GetFromString` with the template parameter set to required numeric type to convert the string value into numeric type `C`. If conversion is impossible, exception `std::range_error` is thrown. Complete reference to this function is available here.

2.2 Exception hierarchy

++Skype exception hierarchy is shown in fig. 1.

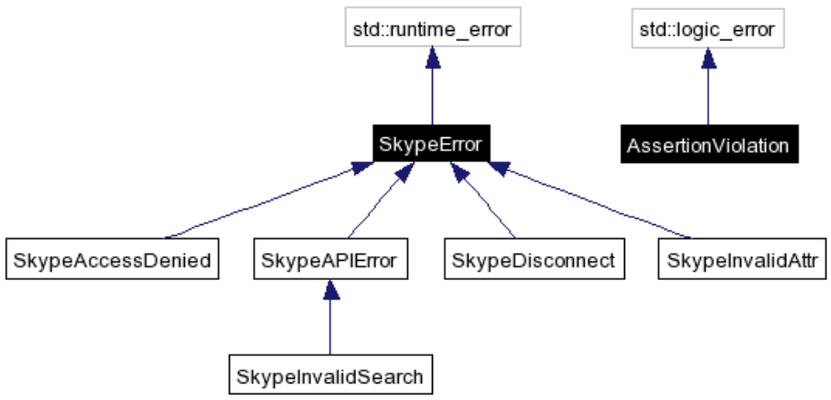

Figure 1: ++Skype exception hierarchy

Exceptions are used by various ++Skype classes to report on critical errors. More details about the ++Skype exception classes are given herebelow:

SkypeError This class is a parent of all ++Skype exception classes (with the only notable exception — class `AssertionViolation`). To catch any runtime error caused by Skype use the reference of type `SkypeError&`.

SkypeAccessDenied, SkypeDisconnect Exceptions of this type are thrown by `SkypeConnection` and `RawSkypeConnection` classes if it was impossible to connect to Skype or if the connection to Skype was broken.

SkypeInvalidAttr Exceptions of this type are thrown by high-level ++Skype components if it was impossible to parse Skype event.

SkypeAPIError This class is a parent of all ++Skype exception classes constructed from Skype API error response (there is only one such class in the current version of ++Skype — `SkypeInvalidSearch`).

SkypeInvalidSearch Exceptions of this type are thrown by high-level components of the library (actually, object managers) if it has got the skype error in response to search query.

AssertionViolation Exceptions of this type are thrown by `Assert` macros. The `Assert` behavior is described here in section 2.1.2.

2.3 Low-level components

2.3.1 RawSkypeConnection template

`RawSkypeConnection` is a template with the only parameter of type `OS_t` — operation system type. `OS_t` is declared in `Cfg.h` library header:

typedef enum { os_win, os_nix } OS_t;

Generic definition of `RawSkypeConnection` leads to a compilation error. There are two `RawSkypeConnection` specializations — for template parameter value `os_win` and for template parameter value `os_nix`. The interfaces of these specializations are very similar. Look at the specialization for `os_win` template parameter value (MS Windows™ version of the `RawSkypeConnection`). Its public member functions are:

```
RawSkypeConnection(const RawNotify_t& _cbk,
                   const std::string& _app_name,
                   LogFunc_t _log_func = DummyLogger);
void Attach(int _timeout = 3000);
bool IsAttached(void);
void Detach();
bool operator()();
void EnableIds();
void DisableIds();
bool IsIdsEnabled() const;
int LastNum() const;
```

To connect to Skype, call `Attach` function, to disconnect from it, call `Detach` function. To check the connection status, use `IsAttached` function. The most interesting argument of the constructor is `_cbk` of type `RawNotify_t`. `RawNotify_t` is declared as functor [1, ch. 5] returning `void` with the only argument of type `const std::string&`:

```
typedef
Loki::Functor<void, LOKI_TYPELIST_1(const std::string&)> RawNotify_t;
```

The functor of this type (provided in the constructor) will be called by the instance of the `RawSkypeConnection` upon receiving any message (notification, error or response to the command) from Skype. The Skype message is the argument of the functor. To initiate processing of incoming Skype messages, call `operator()()` operator[2].

Operators `operator<<` are defined in the `RawSkypeConnection` for integer and boolean data types, STL C++ and null-terminated C-strings. To send message (command) to Skype use these operators and `flush` member function or `flush` manipulator. An example of sending a command is given below (suppose, `sk` variable is of type `RawSkypeConnection`):

```
sk << "PING";
sk.flush();
sk << "SET␣AEC␣" << true << flush;
bool r = GetAgcFromConfig();
sk << "SET␣AGC␣" << r << flush;
```

A useful feature of Skype API is the concept of command identifiers. This concept is fully supported by `RawSkypeConnection`. To turn the command identifiers on, call `EnableIds` member function, to turn them off, call `DisableIds` member function, to get the status of the command identifiers usage, call `IsIdsEnabled`. In addition to these member functions, the corresponding manipulators `EnableIds` and `DisableIds` are provided. If the command identifiers are turned on, each message sent to Skype is prefixed by the automatically generated command number. To get the number of the last command sent, call LastNum. You can use this number as a key to associate an incoming Skype message (response to the command) with the command which has been sent to Skype. SkypeConnection class fulfils this task in this way. By default (upon `RawSkypeConnection` instance creation) the command identifiers are turned on.

We have discussed the usage of MS Windows™ `RawSkypeConnection` specialization. Linux specialization usage is the same, the signature of `Attach` function differs only. The complete reference to the linux specialization is presented here.

Uff-ff. The last item in this section. To declare `RawSkypeConnection` instance use type `RawSkypeConnection<>`. C++ compiler automatically detects

[2]This operator is implemented as an empty function in MS Windows™ specialization of `RawSkypeConnection` and performs the incoming message processing in the linux specialization only. But it is a good platform-independent software design practice to call this operator somewhere in the execution flow on a regular basis. Windows applications should contain the message loop. Incoming Skype messages are processed there.

operation system it runs under and the correct `RawSkypeConnection` specialization is used.

2.3.2 SkypeConnection class

`RawSkypeConnection` template encapsulates all the low-level Skype connection functionality. However, its interface is very limited. `SkypeConnection` inherits `RawSkypeConnection<OS>` where `OS` is a constant of type `OS_t` (look at section 2.3.1). `OS` value is set by the compiler — it detects the operation system it runs under. In addition to the `RawSkypeConnection` interface, `SkypeConnection` provides the following public member functions:

```
unsigned RegisterCallback(const std::string& _regex,
                          const SimpleHandler_t& _f);
unsigned RegisterCallback(const std::string& _regex,
                          const RegExNotify_t& _f);
void UnRegisterCallback(const std::string& _regex,
                        const unsigned& _id);
```

The following functor types [1, ch. 5] are declared in `SkypeConnection.h` library header:

```
typedef Loki::Functor<void,
  LOKI_TYPELIST_1(const std::string&)> SimpleHandler_t;

typedef Loki::Functor<void,
  LOKI_TYPELIST_1(const boost::cmatch&)> RegExNotify_t;

typedef Loki::Functor<void,
  LOKI_TYPELIST_3(const int&,
                  const std::string&,
                  const std::string&)> ErrorHandler_t;
```

The first two functor types — `SimpleHandler_t` and `RegExNotify_t` are used with `RegisterCallback` member functions. The functor of type `ErrorHandler_t` is used as the argument of `Flush` manipulator (represented below).

The object of type SkypeConnection is listening to the incoming Skype API messages. Upon receipt of a Skype message (notification or command response) it is tested for any registered regular expression [2]. To register a regular expression, call one of the `RegisterCallback` member functions. These functions return the unsigned integer identifier of the registered expression. To cancel the registration, call `UnRegisterCallback` member function. If the incoming message matches the regular expression, the corresponding callback functor (of type `SimpleHandler_t` or `RegExNotify_t`) is called. *This procedure is repeated with every registered regular expression.* If the incoming message matches no registered regular expression, the default handler is called. This handler is provided for the object of type `SkypeConnection` as the second argument of the constructor.

Boost.Regex is used for all regular expression operations. Which version of `RegisterCallback` function is better? It depends on your purposes. If there is no need to extract any data from the message, prefer callback functor of type `SimpleHandler_t,` otherwise choose `RegExNotify_t.` The argument of `SimpleHandler_t` is filled with the incoming message text, the `RegExNotify_t` argument is filled with match results of the regular expression.

Examples of `RegisterCallback` usage are given below (suppose, `sk` is of type `SkypeConnection`):

```
void SimpleHandler(const std::string& msg)
{ std::cout << "SimpleHandler: " << msg << "" << std::endl; };

void RegExHandler(const boost::cmatch& m) {
        std::cout << "RegExHandler: " << m[0] << "" << std::endl;
        for(int i=0;i<m.size();i++) {
                std::cout << " match[" << i << "] = " << m[i] << "";
                std::cout << std::endl;
        }
};
```

```
sk.RegisterCallback("^\\s*CONNSTATUS\\s+ONLINE\\s*$",
                    SimpleHandler_t(SimpleHandler));
sk.RegisterCallback("^\\s*CURRENTUSERHANDLE\\s+([^\\s]+)\\s*$",
                    RegExNotify_t(RegExHandler));
```

`SimpleHandler` function will be called upon receipt of the *"CONNSTATUS ONLINE"* notification from Skype. `RegExHandler` function will be called upon receipt of the notification with the name of the currently logged Skype user.

As you have already seen, `ErrorHandler_t` functor type is declared in the library. What is it? Where is this type used? There are two types of incoming Skype API messages — notifications and command responses. The only way to distinguish between these message types is the usage of command identifiers. If the command identifiers are used, each response to a command is provided with the identifier of the corresponding command. If Skype was unable to process the command, it responds with an error message. If no command identifiers are used, it is impossible to discover the command resulted in error. On the contrary, if they are used, this is not a problem. `SkypeConnection` turn the command identifiers on by default. To turn them off, call `DisableIds` public member function of `SkypeConnection` class. To turn them on, call `EnableIds` member function, to get the status of command identifiers usage, call `IsIdsEnabled`. In addition to these member functions, corresponding manipulators `EnableIds` and `DisableIds` are provided. `Flush` manipulator is defined to associate the command with the response to it. This manipulator is implemented as a structure with the following constructor:

```
Flush(const SimpleHandler_t& _sh,
      const ErrorHandler_t& _eh,
      bool _send_event = true);
```

The first argument is a response handler of type `SimpleHandler_t`. This handler will be called upon receiving the response to the command sent with this `Flush` manipulator. The second argument is an error handler of type `ErrorHandler_t`. It will be called if the command has resulted in error. If the third argument of the `Flush` manipulator is set to true (the default value), after the response handler executing event processing mechanism (described above) is activated — the response is tested for all the registered regular expressions and corresponding functors are called if the response matches the regular expression. If this behavior is not what you need, set the last parameter of `Flush` manipulator to false — in which case the response (or error) handler is called only.

`Flush` usage example (suppose, `sk` is of type `SkypeConnection`):

```
void RespHandler(const std::string& msg)
{ std::cout << "RespHandler:␣"" << msg << """ << std::endl; };

void ErrHandler(const int& err_code,
        const std::string& err_msg,
        const std::string& full_err_msg)
{
        std::cout << "Error_code:␣" << err_code << std::endl;
        std::cout << "Error_message:␣" << err_msg << std::endl;
        std::cout << "Full_error_message:␣" << full_err_msg << std::endl;
};

. . .

sk << "PING" << Flush(RespHandler, ErrHandler);
sk << "INCORRECT_COMMAND" << Flush(RespHandler, ErrHandler);
```

Upon receipt of *"PONG"* (response to the *"PING"* command) `RespHandler` will be called. *"INCORRECT COMMAND"* is not a Skype API command, therefore `ErrHandler` will be executed after sending this command to Skype.

2.4 High-level components

2.4.1 SkypeObject template and its descendants

SkypeObject template is a basis for a variety of high-level ++Skype components. All these components (see table 1) inherit public interface of SkypeObject. A typical inheritance diagram for the descendant of SkypeObject is shown in the fig. 2.

Attributes Each Skype object has its attributes. For example, Skype object of type Privileges has the following attributes: *plSkypeOut, plSkypeIn* and *plVoiceMail*. The attribute names are of enumeration type. Such types are called "attribute types". A special attribute type is defined for each descendant of SkypeObject. To map attribute type values to strings (names of the attributes in Skype API notation) specializations of AttrsDescription template are used. Descendants of SkypeObject, their attribute and attribute description types are listed in the table 1.

Class	Attributes type	Attributes description type
Application	ApplicationAttrs_t	ApplicationAttrsDesc_t
AudioSettings	AudioAttrs_t	AudioAttrsDesc_t
Call	CallAttrs_t	CallAttrsDesc_t
Chat	ChatAttrs_t	ChatAttrsDesc_t
ChatMsg	ChatMsgAttrs_t	ChatMsgAttrsDesc_t
Privileges	PrivAttrs_t	PrivAttrsDesc_t
Profile	ProfileAttrs_t	ProfileAttrsDesc_t
User	UserAttrs_t	UserAttrsDesc_t

Table 1: Descendants of SkypeObject template

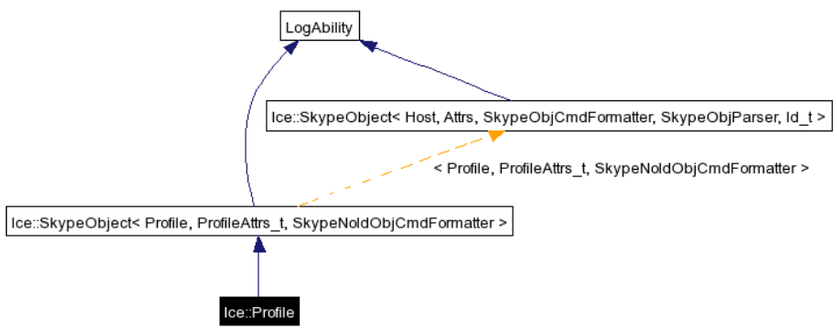

Figure 2: Inheritance diagram for class Profile

The following public members of SkypeObject interface are defined to manage attributes (Attrs is the attribute type):

```
typedef Loki::Functor<void,
LOKI_TYPELIST_5(Host&, const Attrs&, const int&, const std::string&,
                const std::string&)> ObjErrHandler_t;
typedef std::map<Attrs, std::string>::const_iterator  const_iterator ;
typedef AttrsDescription<Attrs> AttrsDescription_t;

const std::string& operator[](const Attrs& _attr) const;

const_iterator  find(const Attrs& _attr) const;

const_iterator  begin() const;
const_iterator  end() const;
```

```
void AskAttr(const Attrs& _attr);
void AskAttr(const Attrs& _attr, const ObjErrHandler_t& _eh);

template <typename T>
void AskForSetAttr(const Attrs& _attr, const T& _value);
template <typename T>
void AskForSetAttr(const Attrs& _attr, const T& _value,
                   const ObjErrHandler_t& _eh);
```

Attribute is stored as `std::pair<Attrs, std::string>`. To get the attribute value, use indexing operator. Call this operator if you are absolutely sure that the attribute value is stored in the object! To iterate through all the attributes stored in the object, use `const_iterator` type. The behavior of `begin(), end()` and `find` member functions is the same as of the corresponding member functions of STL containers.

The value of the attribute is stored as `std::string`. It is easy to convert the string value to other type — use `GetFromString` template function, described in the section 2.1.3.

To get all the attributes stored in the object, iterators can be used in the following way (suppose, `so` is of type `SO_t` and `SO_t` is the descendant of the `SkypeObject`):

```
typedef typename SO_t::const_iterator CI;
const typename SO_t::AttrsDescription_t desc;

for(CI p = so.begin(); p != so.end(); ++p) {
        std::cout << "Obj[" << desc[p->first] << "]=";
        std::cout << p->second << "" << std::endl;
};
```

What is the `AttrsDescription` type? Object of this type maps string names of the attributes into enumeration type and vise versa. Two indexing operators are declared in the public interface of `AttrsDescription` (suppose, `Attrs` is the attribute type) to do this task:

```
const Attrs& operator[](const std::string& _idx) const;
const std::string& operator[](const Attrs& _idx) const;
```

Attributes description types are listed in the table 1. Attribute description type is declared in the public section of each descendant of `SkypeObject` template also — it is called `AttrsDescription_t` there. For example, names `Profile::AttrsDescription_t` and `ProfileAttrsDesc_t` refer to the *same* type — attributes description type for the `ProfileAttrs_t`.

The descendant of `SkypeObject` stores only the attributes it has received from the Skype via API (with the help of `SkypeConnection` instance). If the attribute value has not been reported (by Skype) to the application, it is not stored in the object. To request the attribute value from the Skype, `AskAttr` member functions (one of them) should be called. Note, the behavior of these member functions is asynchronous — they send requests to Skype and don't wait for the answer. That's why it is a wrong expectation to request the attribute value (with the help of indexing operator, for example) immediately after calling `AskAttr` function. But don't worry, there is a special concept in the library to make such things easier — watchers. We will discuss them later in this section.

There is a couple of `AskAttr` member functions — simple (one argument) and advanced (two arguments). Both functions request the attribute value from the Skype. The only difference is the error handling behavior. Skype can return an error message in response to the request. Such errors are handled by the functors of type `ObjErrHandler_t` (see p. 9) which is declared as functor type [1, ch. 5] returning void with five arguments. The *First argument* is the reference to the object which has received the error response, the *second argument* is the attribute name (of enumeration type) the value of which was requested from the Skype, the *third argument* is an integer Skype error code, the *fourth argument* is a human-readable description of the error and the *last argument* is the unmodified error message received from the Skype. If an error occurs, the simple version of `AskAttr` calls the functor of type `ObjErrHandler_t` provided in the object's constructor (see below) while the advanced version calls the functor provided in its second argument.

To set the attribute value (send it to Skype), call one of `AskForSetAttr` member functions. There are two functions with this name — simple (two arguments) and advanced (three arguments). The difference between them is the same as the difference between the pair of `AskAttr` functions, discussed above.

Watchers Values of the attributes are not static. They are changed during the object's lifetime. How to track these changes? How to track the value changes for a specific set of attributes only? *Watchers* is the answer. `StateWatchers` template is defined in `SkypeObj.h` library header. It introduces the following public members (`Host` is the type of `SkypeObject` descendant, `Attrs` is corresponding attribute type):

```
template <class Host, typename Attrs>
class StateWatchers {
public:
    typedef std::set<Attrs> AttrSet_t;

    typedef Loki::Functor<void,
    LOKI_TYPELIST_3(Host&, const Attrs&,
                    const std::string&)> ObjChanged_t;

    unsigned RegisterWatcher(const AttrSet_t& _attrs,
                             const ObjChanged_t& _cbk,
                             const bool& _not_in = false);
    void UnRegisterWatcher(const unsigned& _id);

    void clear();
};
```

Each descendant of `SkypeObject` contains the instance of corresponding `StateWatchers`. It can be obtained via `GetWatchers` member function:

```
typedef StateWatchers<Host, Attrs> Watchers_t;
Watchers_t& GetWatchers();
```

To turn watchers mechanism off or on call `IgnoreWatchers` member function of `SkypeObject` template:

```
void IgnoreWatchers(bool _ignore = true);
```

By default (upon the object creation), the watchers mechanism is turned on.

Let's discuss `StateWatchers` template in-depth. `StateWatchers` contains watchers. To register a watcher, call `RegisterWatcher` member function. It returns the identifier of the registered watcher of unsigned integer type. To cancel the registration of a specific watcher call `UnRegisterWatcher` member function. To cancel the registration of all previously registered watchers, call clear member function.

What is watcher? The watcher is a special object which looks for the changes of attribute values. And if a change happens, it signals about this event with the help of functor of type `ObjChanged_t`. The functor of this type is called (by the watcher) with three arguments — reference to the object which stores the attribute, the name of the changed attribute (of enumeration type) and the new value of the attribute. To define a watcher, call `RegisterWatcher` function. This function accepts three parameters. The first parameter is the set of attribute names. The watcher only looks for the changes of the attribute values the names of which are included (if last parameter is set to false) or not included (if last parameter is set to true) in this set. The second parameter is a functor which is used for signaling.

Looks unclear? OK, doing is better than saying. Just an example (suppose, `au` is of type `AudioSettings`):

```
void ObjChanged(AudioSettings& au, const AudioAttrs_t& attr,
                const std::string& value)
{
        const AudioSettings::AttrsDescription_t desc;
        std::cout << "New_value_of_the_attribute_" << desc[attr];
```

```
        std::cout << "'_is_'" << value << "" << std::endl;
};

typedef AudioSettings::Watchers_t AWatchers_t;
AWatchers_t::AttrSet_t attrs ;
attrs.insert(auAudioOut);
attrs.insert(auAudioIn);
au.GetWatchers().RegisterWatcher(attrs, ObjChanged);
```

Every time when input and output audio devices are changed, `ObjChanged` function is called.

Constructors of `SkypeObject` descendants There are two types of descendants. The descendants of first type have identifiers (integer or string) while the descendants of second type do not. The descendants with identifier are: *Call, Chat, ChatMsg, User* and *Application.* The descendants without identifier are: *AudioSettings, Privileges* and *Profile.* The only difference between their constructors is the additional argument which the constructors of first type descendants accept — identifier. To get the object's identifier, call `GetId` function.

Every `SkypeObject` descendant has two constructors. Look at the constructors of `Privileges` class, for example:

```
Privileges(SkypeConnection& _sc,
           const LogFunc_t& _log_func = DummyLogger);
Privileges(SkypeConnection& _sc, const ObjErrHandler_t& _eh,
           const LogFunc_t& _log_func = DummyLogger);
```

The only difference between these constructors is `_eh` argument. This is the error handler for simple versions of `AskAttr` and `AskForSetAttr` member functions (see p. 10). If constructor without `_eh` parameter is used, dummy (empty) error handler is used in these functions.

Miscellaneous To get the object's identifier, call `GetId` member function. To get the reference to `SkypeConnection` used by the object, call `GetSkype` member function. To get Skype (API) name of the object, use `GetName` member function.

2.4.2 SkypeObjManager template and its descendants

`SkypeObjManager` is a basis for several managers of Skype objects. The following managers are defined in the library:

- Object of type `CallManager` manages objects of type `Call`
- Object of type `ChatManager` manages objects of type `Chat`
- Object of type `ChatMsgManager` manages objects of type `ChatMsg`
- Object of type `UserManager` manages objects of type `User`

The most important functions of the managers are:

Storage of the objects Managed objects (all objects of managed type) are stored in the manager. The manager acts as container of these objects. The objects are automatically put into the manager upon receiving the Skype API notification which contains the new object identifier.

Management of objects lifetime Objects are removed from the container (and memory) if they are no more needed. This behavior prevents the memory leaks.

Notifications on objects creation The conception of listeners is introduced in the managers. The listeners are callback functors [1, ch. 5] which are called upon creation of the new object. All object attributes required for the listener's owner are requested from the Skype before the callback functor has been called.

Object search A search interface is provided. It is possible to search objects which are stored in the manager as well as to send search query to the Skype via Skype API search commands.

We will discuss all these features of managers in detail later in this section.

Listeners Listeners are supported through the `ObjCreated_t` functor type and two member functions — RegisterListener and UnRegisterListener They are declared in the public interface of the `SkypeObjManager` (`ObjHost` is the type of managed objects, `AttrSet_t` type is the set of corresponding object attributes):

```
typedef
Loki::Functor<bool, LOKI_TYPELIST_1(ObjHost&)> ObjCreated_t;

unsigned RegisterListener(const AttrSet_t& _attrs,
                          const ObjCreated_t& _cbk);
void UnRegisterListener(unsigned _id);
```

To register a listener, call `RegisterListener` member function. It returns an unsigned identifier of the listener. To cancel the listener's registration, call `UnRegisterListener` function supplied with this identifier. Each time when a new object is created, the manager checks existence of the attributes required by all the registered listeners (these attributes are provided to the manager via the first argument of the `RegisterListener` function). If all attributes exist in the object, the corresponding listener is called. Otherwise, the missing attributes are requested from the Skype.

The functors of type `ObjCreated_t` return a boolean value. This value is very important! If your program (or program component) does not need the created object (this object will not be used in the program further), it returns false. Otherwise, it returns true. See *object's lifetime management* below for details.

Listeners usage example (`cm` is of type `CallManager`):

```
bool NewCallWatcher(Call& call) {
        bool t = !call[clType].compare("OUTGOING_P2P");
        t = t || (!call[clType].compare("OUTGOING_PSTN"));
        if(t) {
                std::cout << "Outgoing call to: '";
                std::cout << call[clPartnerHandle] << "'" << std::endl;
        };
        return t;
};
. . .
// The same as: std::set<CallAttrs_t> attrs;
CallManager::AttrSet_t attrs;
attrs.insert(clType);
attrs.insert(clPartnerHandle);
cm.RegisterListener(attrs, NewCallWatcher);
```

This example handles outgoing (PSTN or P2P) calls only. Upon execution of the callback functor `NewCallWatcher`, the values of `clType` and `clPartnerHandle` attributes of the call are set.

Search interface There are two search interfaces. The first search interface is used to search though objects which are already stored in the manager. This interface is simple and obvious (`ObjHost` is the type of managed objects, `Id_t` is the type of their identifiers):

```
typedef Loki::SmartPtr<ObjHost,
                       Loki::RefCounted, Loki::AllowConversion> ObjPtr_t;
ObjPtr_t find(const Id_t& _id, bool _want_to_own = false);
```

`find` member function returns a smart pointer [1, ch. 7] of type `ObjPtr_t`. To use this interface, the caller has to know the value of the object's identifier and the object should be stored in the manager.

OK. But what about more complex Skype search queries? There is yet another search interface. Yes, more complex but more powerful!

To initiate the search procedure, `Search` member function should be called:

```
typedef Loki::Functor<bool,
    LOKI_TYPELIST_1(const std::vector<ObjType_t*>&)> SearchResult_t;

void Search(const SearchTypes& _what, const SearchResult_t& _cbk,
            const AttrSet_t& _attrs, const std::string& _param = "");
```

Upon completion of the search, a callback functor of type `SearchResult_t` is executed by the manager. But what are these strange new types — `ObjType_t` and `SearchTypes? ObjType_t` is the type of managed objects. For example, for `CallManager` this type equals to `Call. SearchTypes` is the type of search query. The following types of search queries are defined:

- `CallSearchTypes_t` to be used in the `CallManager` search interface. Possible values are *srCalls, srActiveCalls* and *srMissedCalls.*
- `ChatSearchTypes_t` to be used in the `ChatManager` search interface. Possible values are *srChats, srActiveChats, srBookmarkedChats, srMissedChats* and *srRecentChats.*
- `ChatMsgSearchTypes_t` to be used in the `ChatMsgManager` search interface. Possible values are *scmChatMsgs* and *scmMissedChatMsgs.*
- `UserSearchTypes_t` to be used in the `UserManager` search interface. Possible values are *srUsersWaitingMyAuthorization, srFriends* and *srUsers.*

The third argument of the `Search` function is the set of attributes which are required to be stored in the object (requested from Skype) before calling the callback functor (`_cbk` argument of the `Search` function). The last argument of the call to `Search` is Skype API search parameter. Its meaning depends on the type of the search query. For example, for user search of type `srUsers` the last parameter is a part of username or e-mail to match. And, of course, the search behavior is asynchronous.

Search usage example (`chm` is of type `ChatManager`):

```
bool SearchResult(const std::vector<Chat*>& resp) {
        std::cout << "Active chat search result:" << std::endl;
        typedef std::vector<Chat*>::const_iterator CI;
        for(CI p = resp.begin(); p != resp.end(); ++p) {
                Chat& ch = **p;
                std::cout << "Chat '" << ch[chName];
                std::cout << "' with the members: '";
                std::cout << ch[chMembers] << "'" << std::endl;
        };
        return false;
};

// The same as: std::set<ChatAttrs_t> attrs;
ChatManager::AttrSet_t attrs;
attrs.insert(chMembers);
attrs.insert(chName);
chm.Search(srActiveChats, SearchResult, attrs);
```

This example searches for active chats. Upon completion, the names and members of active chats are printed out. Note, `SearchResult` returns a boolean value (false in the example above). If your program (or program component) is not going to use the objects returned by the search query after callback functor completion, this functor has to return false. Otherwise, return is true from it. See *object lifetime management* below for details.

Object lifetime management Objects of managed types are stored in the managers. These objects are created upon receiving Skype message with unknown object's identifier. But it is not a good idea to store all such Skype

objects in the memory because the computer memory is limited, even nowadays! If the object is not used by the program, the manager removes it from the memory. How is the manager informed that object A is used by the program while object B is not used? There are several ways to inform the manager. The first way is the return value of the listener (see above). The second way is the return value of the search callback function (see above). However, the usual way is to call `ReleaseObj` member function (`ObjHost` is the type of managed objects):

```
void ReleaseObj(const ObjHost& _obj);
```

Let's explain — you got a new object via listener or search callback mechanism. You returned true from the functors because you needed the object(s). You worked with the object. And when everything has been done (for example, the call you were monitoring, was finished) you do not need the object anymore. OK, just call `ReleaseObj` member function!

This function performs reference counting for the object. If the object's usage counter runs up to zero, it marks the object as "released" but does not remove it from the manager. The real dirty work is implemented in the special function `Squeeze:`

```
void Squeeze();
```

This function must not be called from any ++Skype callback functors. This is very important! And it is very important to call this function on a regular basis. Otherwise, memory leaks can occur.

To check the quantity of objects currently stored in the manager, call `Size` member function:

```
unsigned Size() const;
```

2.4.3 Application and AppStream classes

We'll discuss these classes in the future releases of this tutorial. They are used to implement application to application communication.

`Application` reference is available here, `AppStream` reference is available here.

3 Additional ++Skype resources

A very good source of ++Skype information is its documentation. It is available online. You can discuss ++Skype with other users on web forums. And, of course, don't hesitate to contact us!

Up-to-date list of ++Skype resources can be found here.

References

[1] *A. Alexandrescu.* Modern C++ Design (Addison-Wesley, 2001)

[2] *Jeffrey E. F. Friedl* Mastering Regular Expressions, Second Edition (O'Reilly, 2002)