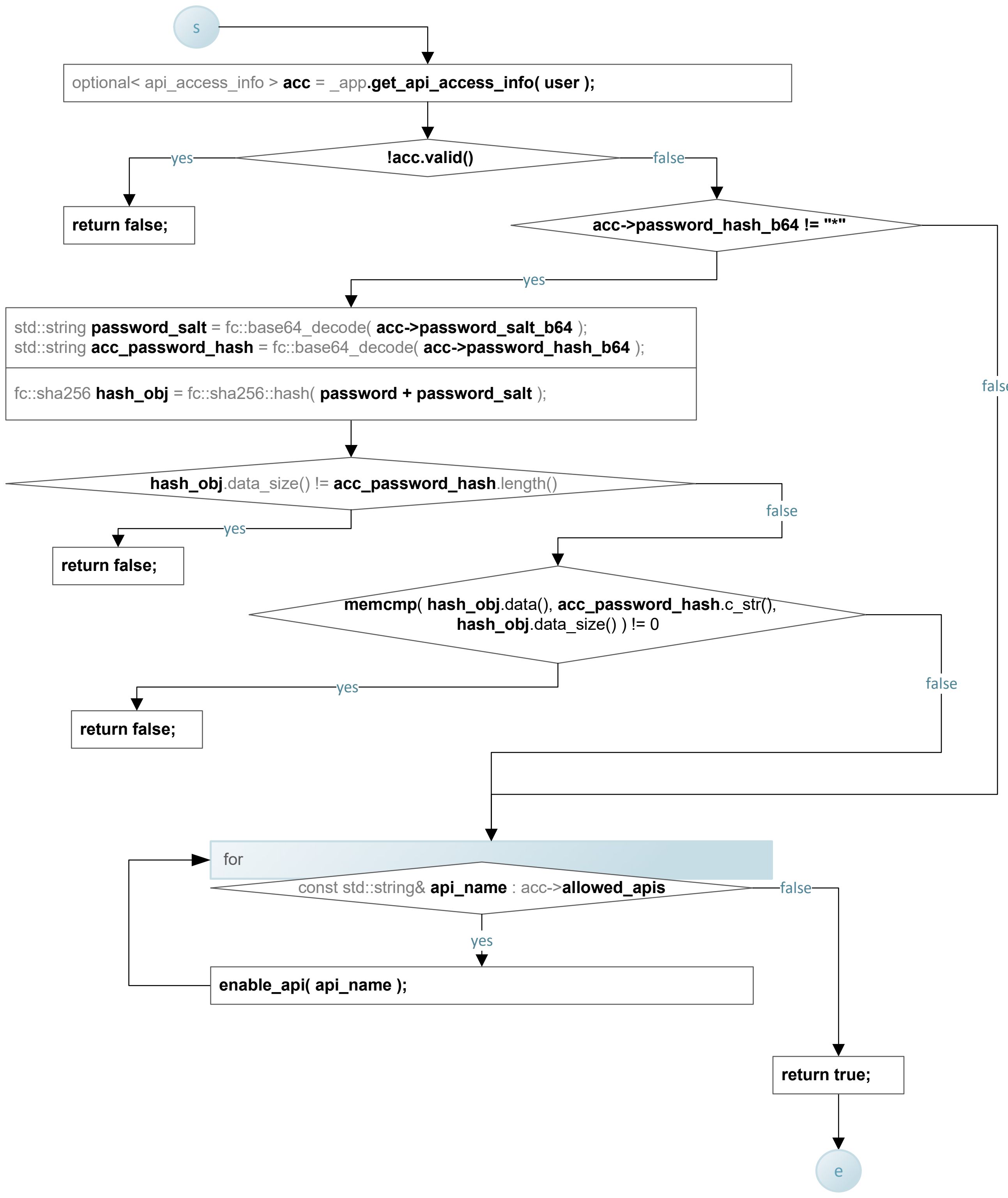
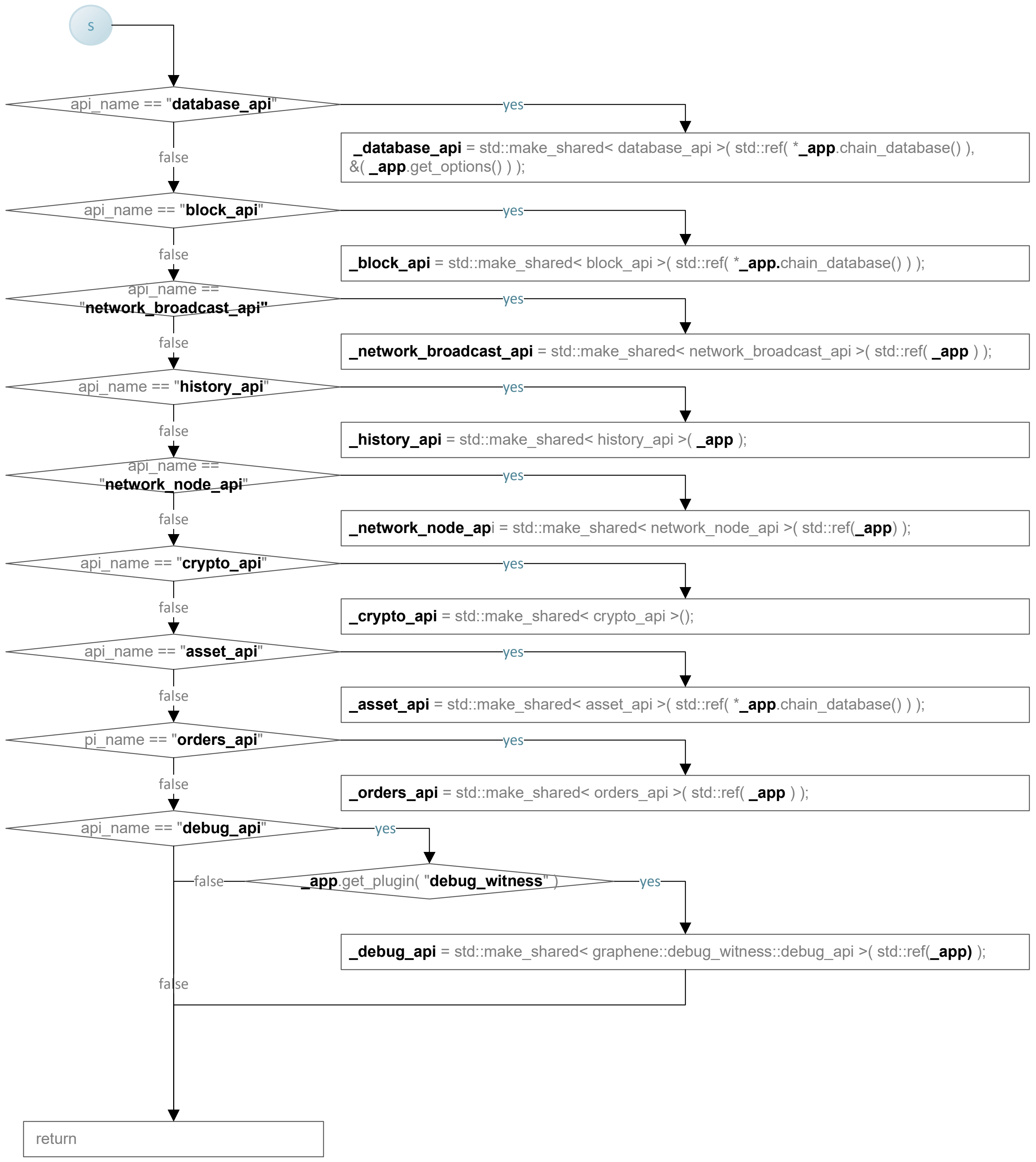
bool login_api::**login**(const string& **user**, const string& **password**)

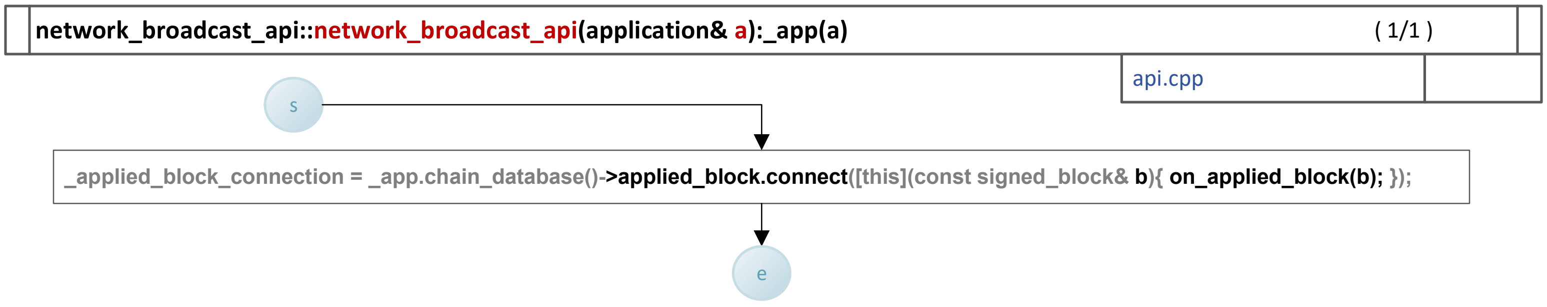
api.cpp


s
optional< api_access_info > acc = _app.get_api_access_info(user);
!acc.valid()
yes
false
return false;
acc->password_hash_b64 != "*"
yes
false
std::string password_salt = fc::base64_decode(acc->password_salt_b64);
std::string acc_password_hash = fc::base64_decode(acc->password_hash_b64);
fc::sha256 hash_obj = fc::sha256::hash(password + password_salt);
hash_obj.data_size() != acc_password_hash.length()
yes
false
return false;
memcmp(hash_obj.data(), acc_password_hash.c_str(), hash_obj.data_size()) != 0
yes
false
return false;
for
const std::string& api_name : acc->allowed_apis
false
yes
enable_api(api_name);
return true;
e

void login_api::**enable_api(** const std::string& **api_name**)

api.cpp

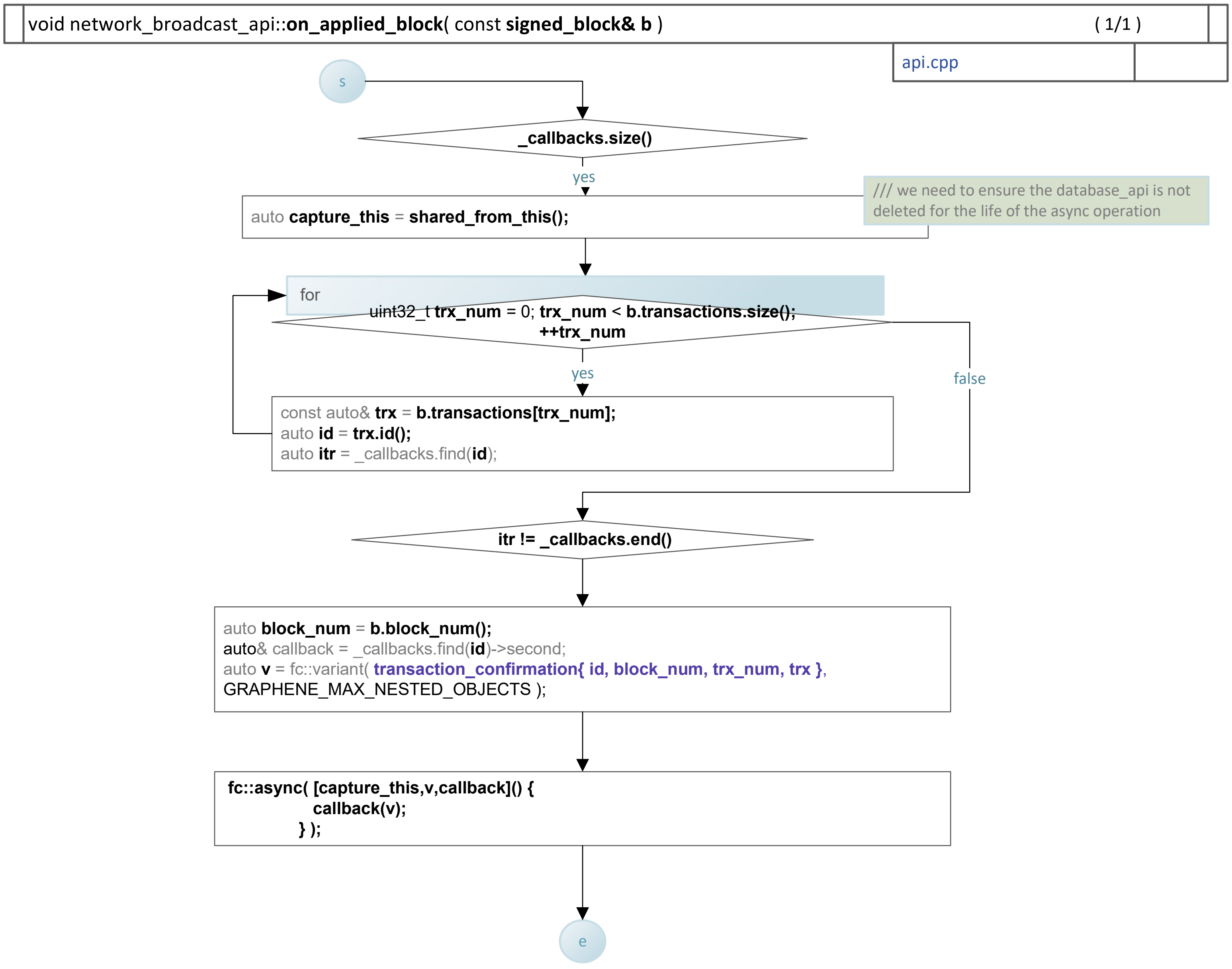

S
api_name == "database_api"
yes
_database_api = std::make_shared< database_api >(std::ref(*_app.chain_database()), &(_app.get_options()));
false
api_name == "block_api"
yes
_block_api = std::make_shared< block_api >(std::ref(*_app.chain_database()));
false
api_name == "network_broadcast_api"
yes
_network_broadcast_api = std::make_shared< network_broadcast_api >(std::ref(_app));
false
api_name == "history_api"
yes
_history_api = std::make_shared< history_api >(_app);
false
api_name == "network_node_api"
yes
_network_node_api = std::make_shared< network_node_api >(std::ref(_app));
false
api_name == "crypto_api"
yes
_crypto_api = std::make_shared< crypto_api >();
false
api_name == "asset_api"
yes
_asset_api = std::make_shared< asset_api >(std::ref(*_app.chain_database()));
false
pi_name == "orders_api"
yes
_orders_api = std::make_shared< orders_api >(std::ref(_app));
false
api_name == "debug_api"
yes
_app.get_plugin("debug_witness")
false
yes
_debug_api = std::make_shared< graphene::debug_witness::debug_api >(std::ref(_app));
false
return



network_broadcast_api::network_broadcast_api(application& a):_app(a) (1/1)

api.cpp

```
_applied_block_connection = _app.chain_database()->applied_block.connect([this](const signed_block& b){ on_applied_block(b); });
```



void network_broadcast_api::**on_applied_block**(const **signed_block& b**) (1/1)

api.cpp

```
_callbacks.size()
```

```
auto capture_this = shared_from_this();
```

/// we need to ensure the database_api is not deleted for the life of the async operation

```
uint32_t trx_num = 0; trx_num < b.transactions.size(); ++trx_num
```

```
const auto& trx = b.transactions[trx_num];
auto id = trx.id();
auto itr = _callbacks.find(id);
```

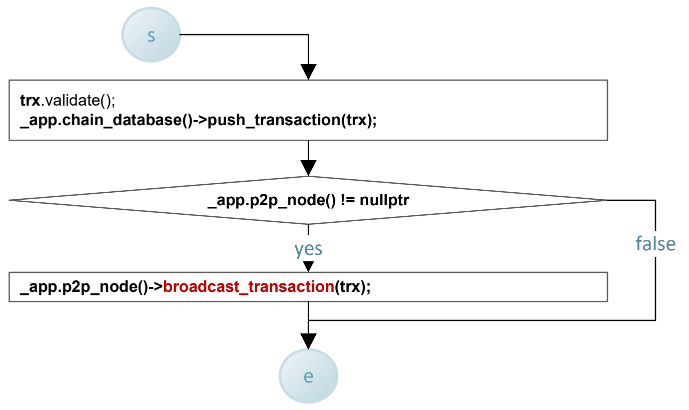
```
itr != _callbacks.end()
```

```
auto block_num = b.block_num();
auto& callback = _callbacks.find(id)->second;
auto v = fc::variant( transaction_confirmation{ id, block_num, trx_num, trx },
GRAPHENE_MAX_NESTED_OBJECTS );
```

```
fc::async( [capture_this,v,callback]() {
             callback(v);
           } );
```

void network_broadcast_api::**broadcast_transaction**(const **signed_transaction& trx**) (1/1)

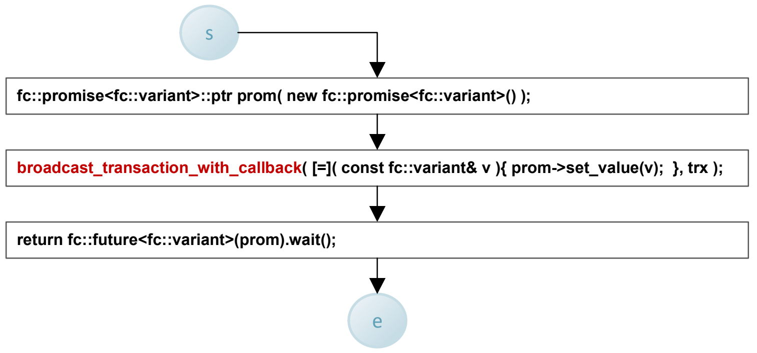
api.cpp



```
trx.validate();
_app.chain_database()->push_transaction(trx);
_app.p2p_node() != nullptr
_app.p2p_node()->broadcast_transaction(trx);
```

fc::variant network_broadcast_api::**broadcast_transaction_synchronous(**const **signed_transaction& trx**) (1/1)

api.cpp



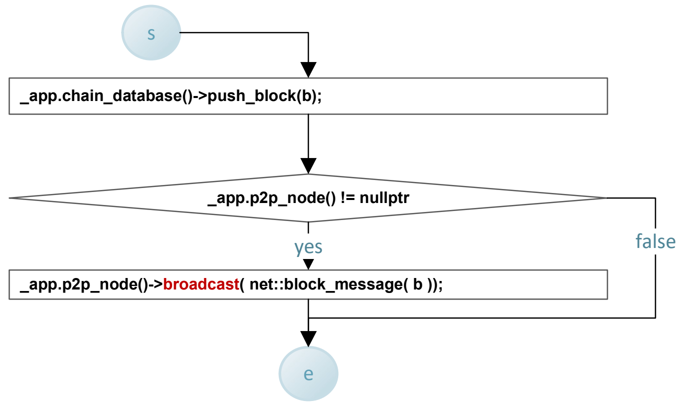
```
fc::promise<fc::variant>::ptr prom( new fc::promise<fc::variant>() );
broadcast_transaction_with_callback( [=]( const fc::variant& v ){ prom->set_value(v);  }, trx );
return fc::future<fc::variant>(prom).wait();
```

fc::variant broadcast_transaction_synchronous(const signed_transaction &trx)

this version of broadcast transaction registers a callback method that will be called when the transaction is included into a block. The callback method includes the transaction id, block number, and transaction number in the block.

void network_broadcast_api::**broadcast_block**(const **signed_block& b**) (1/1)

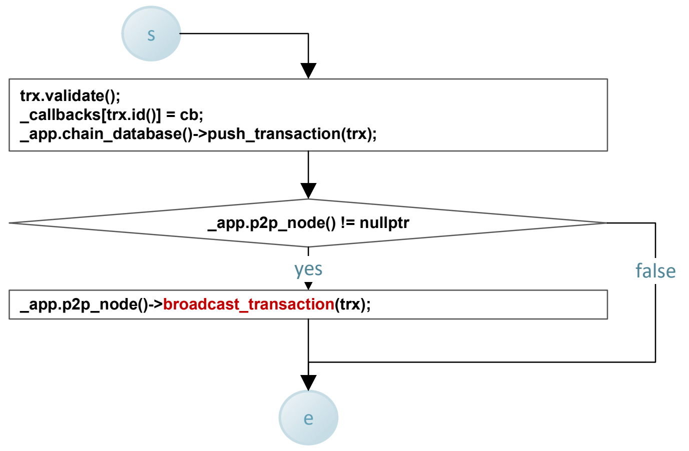
api.cpp



```
_app.chain_database()->push_block(b);
_app.p2p_node() != nullptr
_app.p2p_node()->broadcast( net::block_message( b ));
```

void network_broadcast_api::**broadcast_transaction_with_callback**(confirmation_callback **cb**, const **signed_transaction& trx**)

api.cpp



```
trx.validate();
_callbacks[trx.id()] = cb;
_app.chain_database()->push_transaction(trx);
_app.p2p_node() != nullptr
_app.p2p_node()->broadcast_transaction(trx);
```

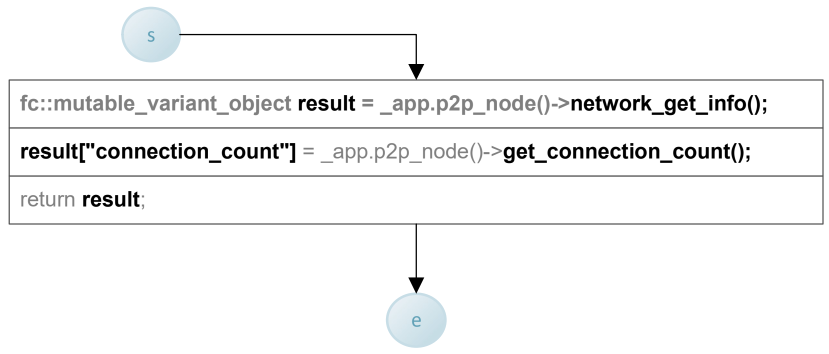
processed_transaction push_transaction(const signed_transaction &trx, uint32_t skip = skip_nothing)

Attempts to push the transaction into the pending queue

When called to push a locally generated transaction, set the skip_block_size_check bit on the skip argument. This will allow the transaction to be pushed even if it causes the pending block size to exceed the maximum block size. Although the transaction will probably not propagate further now, as the peers are likely to have their pending queues full as well, it will be kept in the queue to be propagated later when a new block flushes out the pending queues.

fc::variant_object network_node_api::**get_info()** const (1/1)

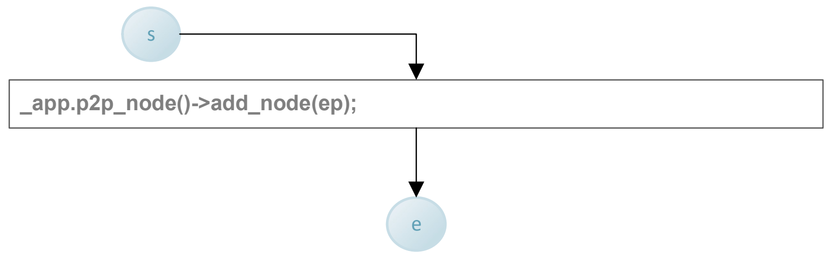
api.cpp



```
fc::mutable_variant_object result = _app.p2p_node()->network_get_info();
result["connection_count"] = _app.p2p_node()->get_connection_count();
return result;
```

void network_node_api::**add_node**(const fc::ip::endpoint& **ep**) (1/1)

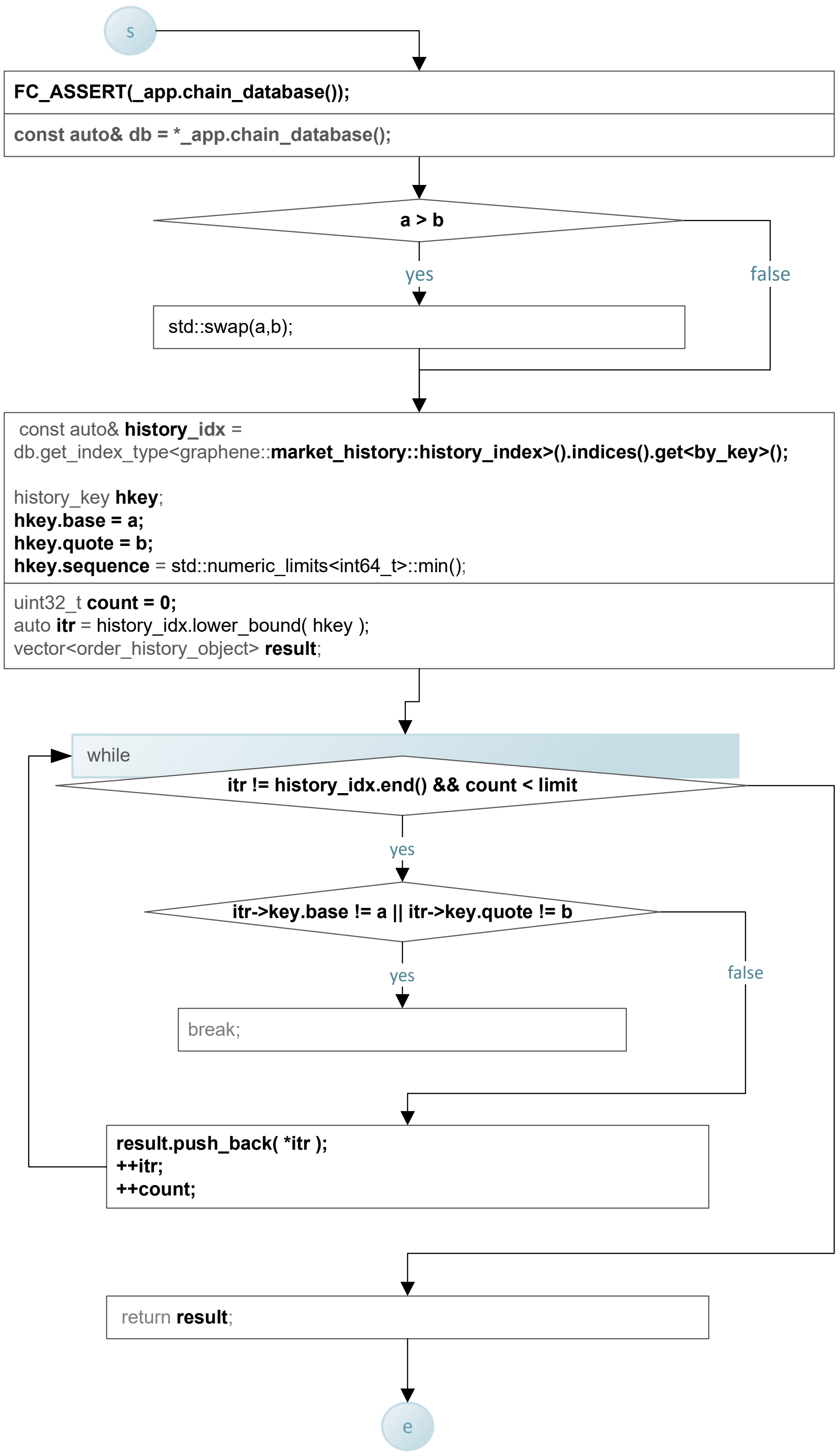
api.cpp



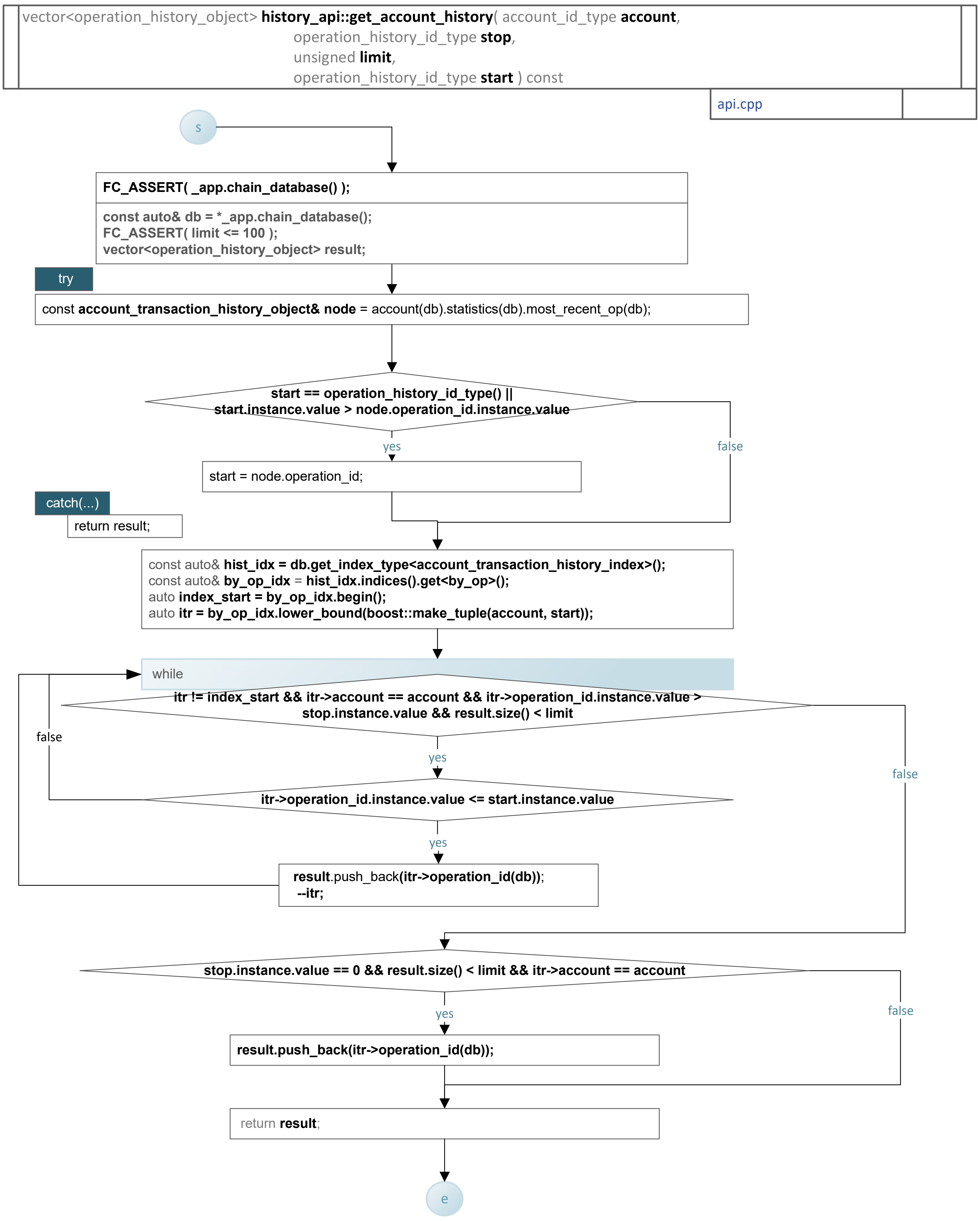
```
_app.p2p_node()->add_node(ep);
```

vector<order_history_object> history_api::**get_fill_order_history**(asset_id_type **a,** asset_id_type **b**, uint32_t **limit**)const

api.cpp



s
FC_ASSERT(_app.chain_database());
const auto& db = *_app.chain_database();
a > b
yes
false
std::swap(a,b);
const auto& history_idx = db.get_index_type<graphene::market_history::history_index>().indices().get<by_key>();
history_key hkey;
hkey.base = a;
hkey.quote = b;
hkey.sequence = std::numeric_limits<int64_t>::min();
uint32_t count = 0;
auto itr = history_idx.lower_bound(hkey);
vector<order_history_object> result;
while
itr != history_idx.end() && count < limit
yes
itr->key.base != a || itr->key.quote != b
yes
false
break;
result.push_back(*itr);
++itr;
++count;
return result;
e



```
vector<operation_history_object> history_api::get_account_history( account_id_type account,
                                                                   operation_history_id_type stop,
                                                                   unsigned limit,
                                                                   operation_history_id_type start ) const
```

api.cpp

```
FC_ASSERT( _app.chain_database() );

const auto& db = *_app.chain_database();
FC_ASSERT( limit <= 100 );
vector<operation_history_object> result;
```

try

```
const account_transaction_history_object& node = account(db).statistics(db).most_recent_op(db);
```

```
start == operation_history_id_type() ||
start.instance.value > node.operation_id.instance.value
```

yes

```
start = node.operation_id;
```

false

catch(...)

```
return result;
```

```
const auto& hist_idx = db.get_index_type<account_transaction_history_index>();
const auto& by_op_idx = hist_idx.indices().get<by_op>();
auto index_start = by_op_idx.begin();
auto itr = by_op_idx.lower_bound(boost::make_tuple(account, start));
```

while

```
itr != index_start && itr->account == account && itr->operation_id.instance.value >
stop.instance.value && result.size() < limit
```

false

yes

false

```
itr->operation_id.instance.value <= start.instance.value
```

yes

```
result.push_back(itr->operation_id(db));
 --itr;
```

```
stop.instance.value == 0 && result.size() < limit && itr->account == account
```

yes

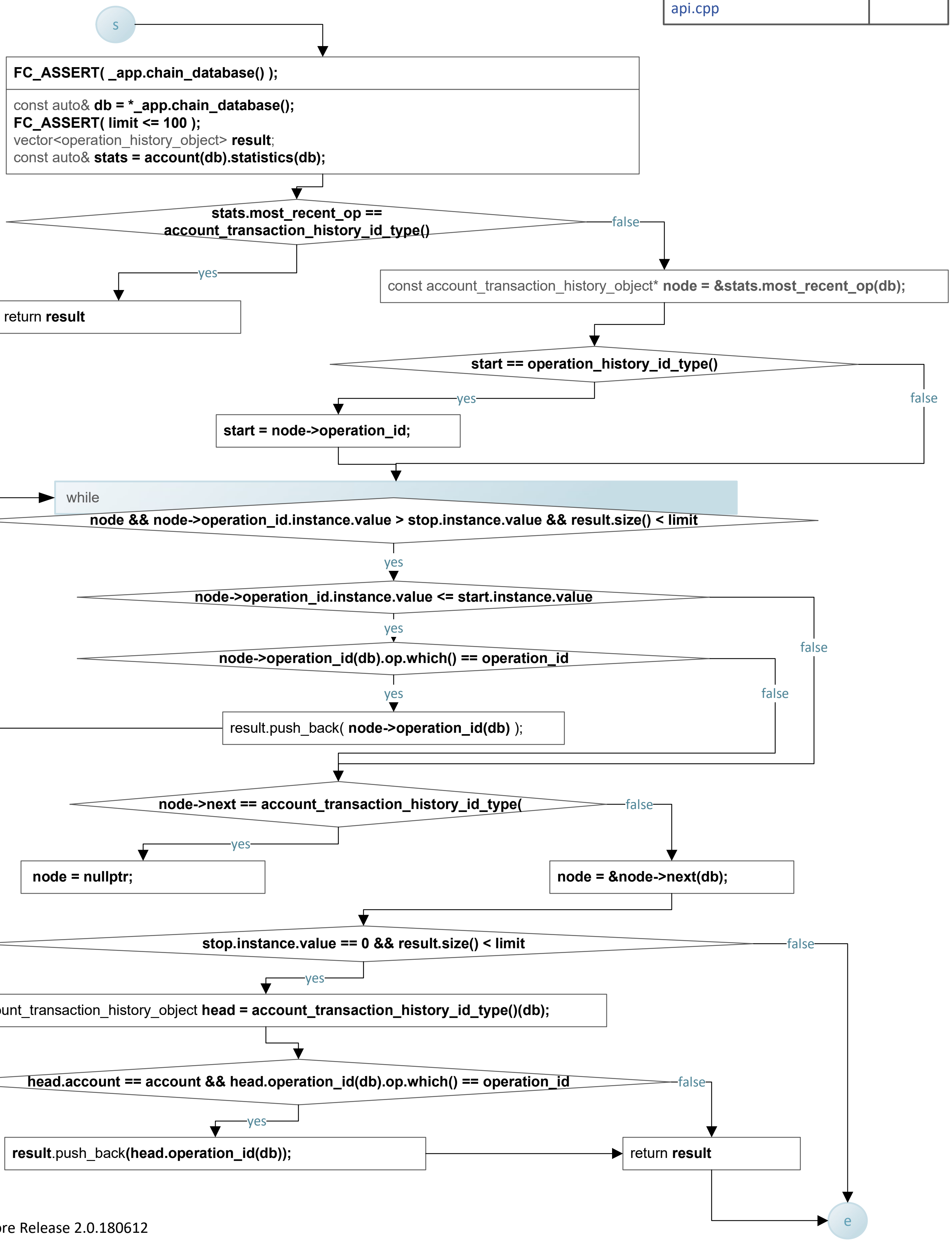
false

```
result.push_back(itr->operation_id(db));
```

```
return result;
```

```
vector<operation_history_object> history_api::get_account_history_operations( account_id_type account,
                                                                              int operation_id,
                                                                              operation_history_id_type start,
                                                                              operation_history_id_type stop,
                                                                              unsigned limit) const
```

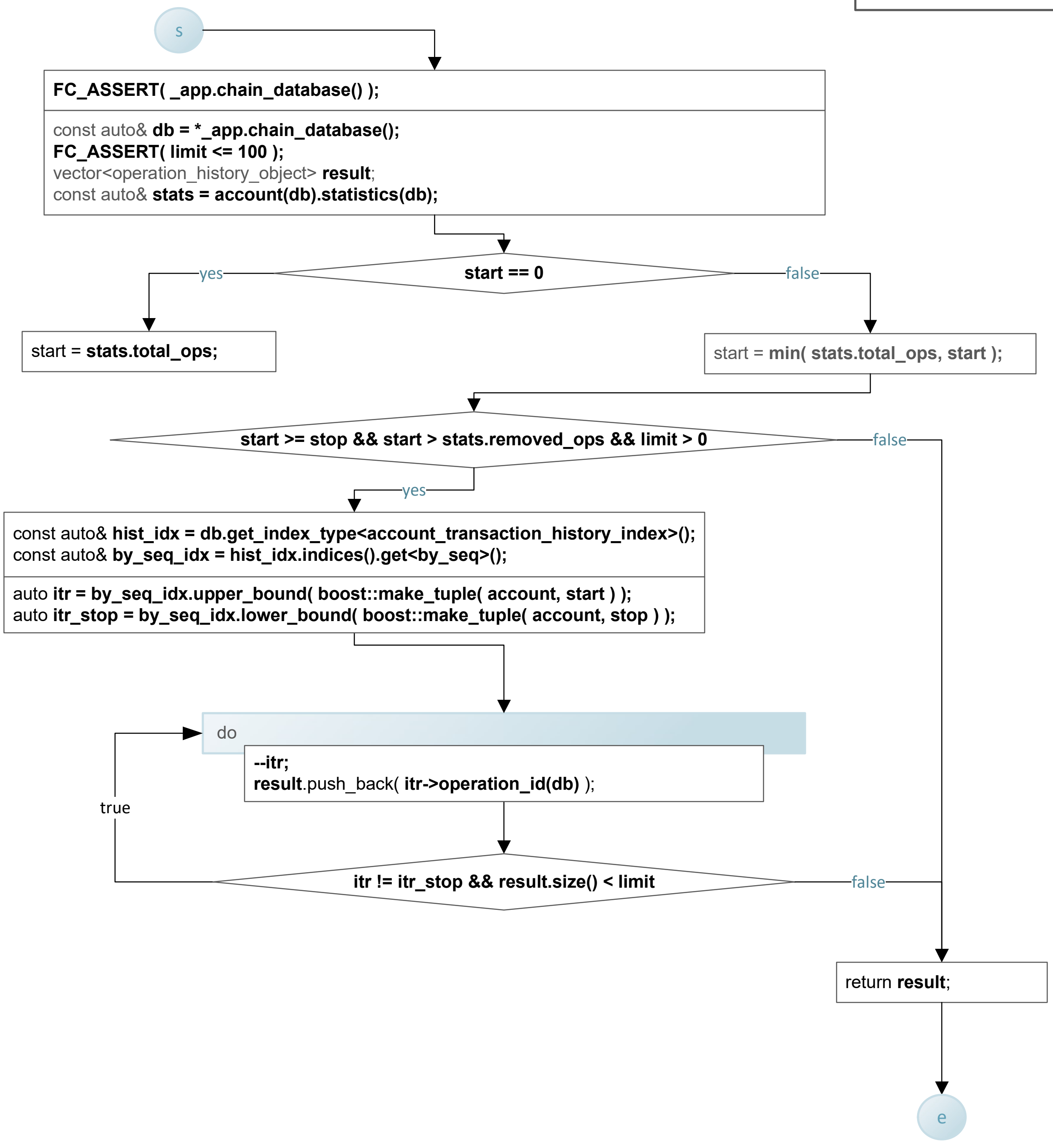
api.cpp



```
FC_ASSERT( _app.chain_database() );

const auto& db = *_app.chain_database();
FC_ASSERT( limit <= 100 );
vector<operation_history_object> result;
const auto& stats = account(db).statistics(db);

stats.most_recent_op == account_transaction_history_id_type()
    yes: return result
    false: const account_transaction_history_object* node = &stats.most_recent_op(db);

start == operation_history_id_type()
    yes: start = node->operation_id;

while
node && node->operation_id.instance.value > stop.instance.value && result.size() < limit
    yes:
    node->operation_id.instance.value <= start.instance.value
        yes:
        node->operation_id(db).op.which() == operation_id
            yes: result.push_back( node->operation_id(db) );

node->next == account_transaction_history_id_type(
    yes: node = nullptr;
    false: node = &node->next(db);

stop.instance.value == 0 && result.size() < limit
    yes: const account_transaction_history_object head = account_transaction_history_id_type()(db);

head.account == account && head.operation_id(db).op.which() == operation_id
    yes: result.push_back(head.operation_id(db));
    false: return result
```

```
vector<operation_history_object> history_api::get_relative_account_history( account_id_type account,
                                                                           uint32_t stop,
                                                                           unsigned limit,
                                                                           uint32_t start) const
```

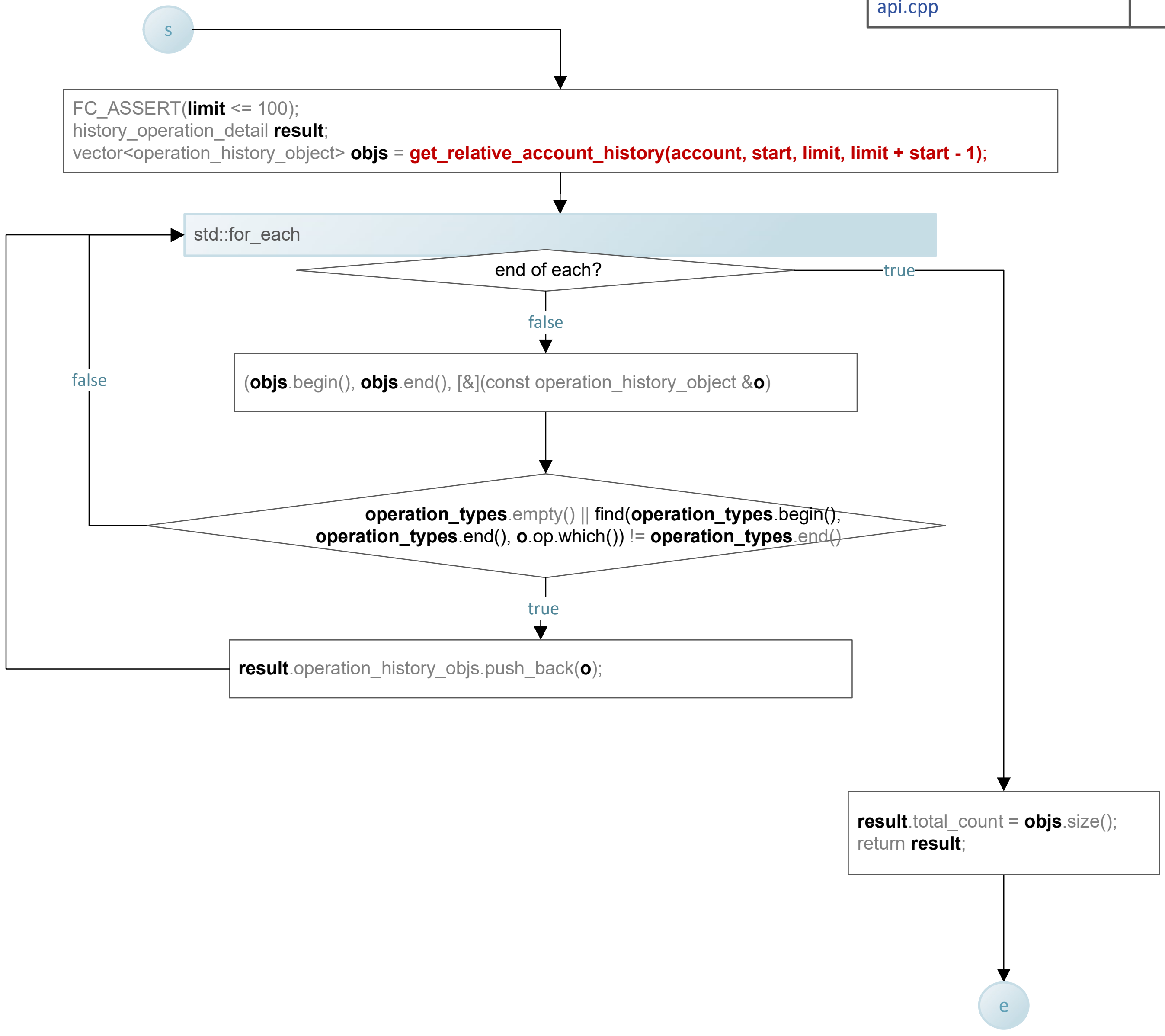
api.cpp



```
FC_ASSERT( _app.chain_database() );

const auto& db = *_app.chain_database();
FC_ASSERT( limit <= 100 );
vector<operation_history_object> result;
const auto& stats = account(db).statistics(db);

start == 0

start = stats.total_ops;

start = min( stats.total_ops, start );

start >= stop && start > stats.removed_ops && limit > 0

const auto& hist_idx = db.get_index_type<account_transaction_history_index>();
const auto& by_seq_idx = hist_idx.indices().get<by_seq>();

auto itr = by_seq_idx.upper_bound( boost::make_tuple( account, start ) );
auto itr_stop = by_seq_idx.lower_bound( boost::make_tuple( account, stop ) );

do

--itr;
result.push_back( itr->operation_id(db) );

itr != itr_stop && result.size() < limit

return result;
```

history_operation_detail history_api::**get_account_history_by_operations**
(account_id_type **account**, vector<uint16_t> **operation_types**, uint32_t **start**, unsigned **limit**)

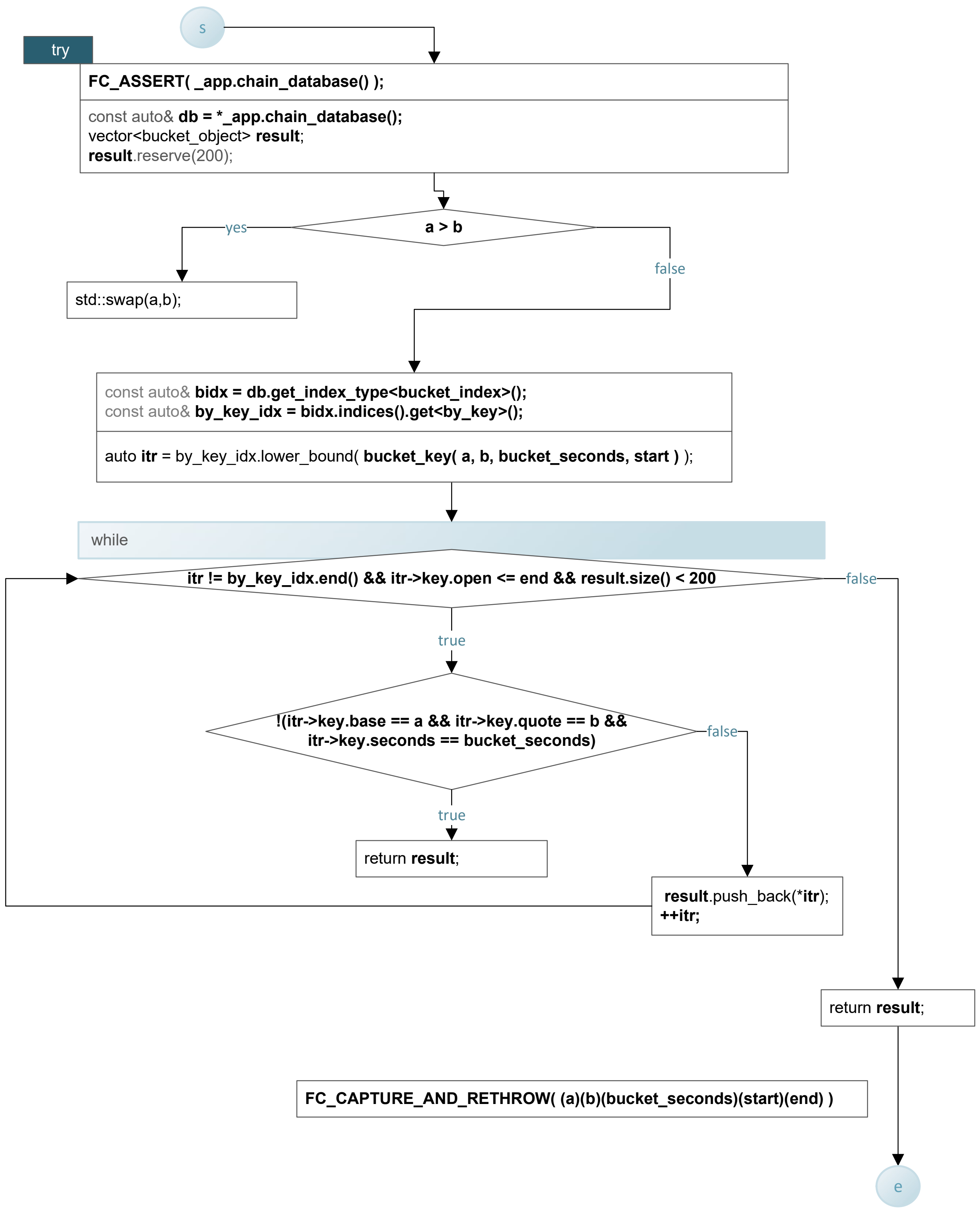
api.cpp



s
FC_ASSERT(limit <= 100);
history_operation_detail result;
vector<operation_history_object> objs = get_relative_account_history(account, start, limit, limit + start - 1);
std::for_each
end of each?
true
false
(objs.begin(), objs.end(), [&](const operation_history_object &o)
false
operation_types.empty() || find(operation_types.begin(), operation_types.end(), o.op.which()) != operation_types.end()
true
result.operation_history_objs.push_back(o);
result.total_count = objs.size();
return result;
e

vector<bucket_object> history_api::**get_market_history**(asset_id_type **a**, asset_id_type **b**,
uint32_t **bucket_seconds**, fc::time_point_sec **start**, fc::time_point_sec **end**)const

api.cpp



```
try
FC_ASSERT( _app.chain_database() );
const auto& db = *_app.chain_database();
vector<bucket_object> result;
result.reserve(200);
a > b
std::swap(a,b);
const auto& bidx = db.get_index_type<bucket_index>();
const auto& by_key_idx = bidx.indices().get<by_key>();
auto itr = by_key_idx.lower_bound( bucket_key( a, b, bucket_seconds, start ) );
while
itr != by_key_idx.end() && itr->key.open <= end && result.size() < 200
!(itr->key.base == a && itr->key.quote == b &&
itr->key.seconds == bucket_seconds)
return result;
result.push_back(*itr);
++itr;
return result;
FC_CAPTURE_AND_RETHROW( (a)(b)(bucket_seconds)(start)(end) )
```