PostgreSQL

Part A. Introduction to PostgreSQL

Terminologies :

• `postgresql`	PostgreSQL server executable, run as a daemon
• `postgres`	linux account (auto-created when we install PostgreSQL)
• `postgres`	PostgreSQL account by default
• `postgres`	PostgreSQL database by default
• `psql`	PostgreSQL client executable, run as a process

Differences between daemon and process are :

- daemon's parent is `init`, i.e. parent of all processes
- daemon is not connected to a terminal
- daemon scripts can be found inside folder `/etc/init.d` (including `postgresql`)
- started by `service` command, with various options `start`, `stop` and `restart`, for example `>> sudo service postgresql start`

Difference between `su` command and `sudo` command :

`>> su`	substitute-user as root, need to enter password of root
`>> su peter`	substitute-user as `peter`, need to enter password of `peter`
`>> sudo command arg`	substitute-user as root to run one command, need to enter password of `dick.chow`
`>> sudo -u peter command arg`	substitute-user as `peter` to run one command, need to enter password of `dick.chow`
`>> sudo -iu peter`	substitute-user as `peter` to run interactively, need to enter password of `dick.chow`

Difference between authentication and authorization :

- authentication means confirmation of a user's identity, i.e. about login
- authorization means allowing a user an access to system, i.e. about access right

Difference between database and table :

- one database has multiple tables, govered by a high-level schematic
- one table is a sequence of row-data

In order to start PostgreSQL, we need :

- to start PostgreSQL server `postgresql` as a daemon
- to start PostgreSQL client `psql` as a process
- in order to start `psql`, we need to connect :

- to a specific PostgreSQL database (for example `dbname`) ***with anyname***
- from a specific PostgreSQL account (for example `peter`) ***with the same name*** as current linux account (for example `peter`)
- by running command as of linux account `peter`

```
>> psql -h hostname -U peter -d dbnam
```

► option `-h` is for PostgreSQL hostname
option `-U` is for PostgreSQL account *(not linux account, suppose you are running* `psql` *with appropriate linux account)*
option `-d` is for PostgreSQL database

- if PostgreSQL hostname is not specified, the default PostgreSQL host is local host
- if PostgreSQL account is not specified, the default PostgreSQL account is the same as current linux account
- if PostgreSQL database is not specified, the default PostgreSQL database is the same as PostgreSQL account
- hence we have 2 approaches to start `psql` :

```
// method 1 : one-liner approach
>> sudo -u peter psql -U peter -d dbname

// or simply ...
>> sudo -u peter psql -d dbname

// or simply ...
>> sudo -u peter psql
```

```
// method 2 : two-steps approach
>> sudo -iu peter
>> psql -U  peter -d dbname
// or simply ...
>> sudo -iu peter
>> psql -d  dbname
// or simply ...
>> sudo -iu peter
>> psql
```

```
NOTE :
-u is for sudo
-U is for psql
```

Part B. Create PostgreSQL account and PostgreSQL database

During installation of PostgreSQL, the following are created as default :

- linux account `postgres`
- PostgreSQL account `postgres`
- PostgreSQL database `postgres`, `template0` and `template1`
- ► Thus when we login `psql` for the first time, we can make use of the above *"linux accounts - PostgreSQL account"* pair.

Create custom account and database

Linux account can be created by linux command `adduser` while PostgreSQL account (like `peter`) and PostgreSQL database (like `dbname`) can be created using account `postgres` in two approaches :

1. `sudo` as linux account `postgres`, run ***PostgreSQL utility*** (binary) such as `createuser` and `createdb`
2. login `psql` as account `postgres`, run ***PostgreSQL command*** such as `CREATE USER` and `CREATE DATABASE`

Difference between utility and command :

- ***PostgreSQL utility*** is just a wrapper of ***PostgreSQL command*** (or vice versa?)
- ***PostgreSQL utility*** is run in terminal, it is case-sensitive, as it is executable
- ***PostgreSQL command*** is run in `psql` client, it is case-insensitive, ***MUST*** be terminated with semicolon

About authentication

After installation of PostgreSQL, there is a config file `pg_hba.conf` for management of authenication :

```
>> find -name pg_hba* 2>/dev/null
/etc/postgresql/10/main/pg_hba.conf                    // I have installed two psql versions
/etc/postgresql/13/main/pg_hba.conf
>> sudo cat /etc/postgresql/10/main/pg_hba.conf
...
```

Inside `pg_hba.conf`, there are 3 types of authenication :

- `peer` requires PostgreSQL account to be the same as linux account, and no password is needed
- `MD5` does not require PostgreSQL account to be the same as linux account, and password is necessary
- `trust` does not require any password, it simply trusts all local connections

By default, `peer` is used, therefore we need to :

`>> sudo -u peter psql -U peter -d dbname` to access database through `psql` client ***OR***

`>> sudo -u peter ./my_exec_using_C_API` to access database through `C-API`

Let's start from scratch.

```
// Step 0. Installation
>> sudo apt update
>> sudo apt install postgresql postgresql-contrib       // install 2 things, the former is the core, the latter is the extension
>> which psql
>> psql --version


// Step 1. Start postgresql server
>> sudo service postgresql start
* Starting PostgreSQL 10 database server
>> ps -aux | grep postgres
multiple postgres processes are displayed, such as
- checkpointer process
- collector process
- launcher process
- logical process
- writer process
- etc ...


// Step 2. Start psql client (with the only default account) NOTE : -u is for sudo, -U is for psql
method 1 : one-liner approach
>> sudo -u postgres psql -U postgres -d postgres        OR
>> sudo -u postgres psql -d postgres                    OR
>> sudo -u postgres psql
method 2 : two-steps approach
>> sudo -iu postgres;     psql -U postgres -d postgres  OR
>> sudo -iu postgres;     psql -d postgres              OR
>> sudo -iu postgres;     psql                          OR

// All above methods bring you to psql, with active db-name as prompt ... lets try different psql commands ...
postgres=#
postgres=# \conninfo                                    // check connection info : current username, current database and port num
postgres=# \c db001                                     // connect to database db001
postgres=# \l                                           // list all databases
postgres=# \du                                          // list all users
postgres=# \dt                                          // list all tables    in current database
postgres=# \ds                                          // list all sequences in current database
postgres=# \d                                           // list all relations in current database (i.e. tables + sequences)
postgres=# \d table001                                  // list all columns of table table001 in current database
postgres=# \h                                           // list all commands
postgres=# \q                                           // quit client psql
>>


// Step 3. Create linux account
>> less /etc/passwd                                     // list all users
>> sudo adduser peter
>> sudo deluser dummy
>> sudo passwd peter


// Step 4. Create postgreSQL account and database
method 1 : using psql utility in terminal (remember to use linux account postgres instead of your own)
>> sudo -iu postgres                                    // alternatively, by one-liner
>> createuser peter                                     >> sudo -u postgres createuser peter
>> createdb book_db                                     >> sudo -u postgres createdb bookdb
>> createdb document_db                                 >> ...
>> createdb download_db
>> dropdb   download_db
>> sudo -u peter psql -U peter -d book_db               // connect to newly-created-database using newly-created-account
book_db=>
book_db=> \c document_db                                // connect to another newly-created-database
document_db=>
document_db=> \q


method 2 : using psql command in psql
>> sudo -iu postgres
>> psql -U postgres -d template1
template1=# CREATE USER peter WITH PASSWORD '12qwasZX';
template1=# CREATE DATABASE book_db;
template1=# CREATE DATABASE document_db;
template1=# CREATE DATABASE download_db;
template1=# DROP   DATABASE download_db;
template1=# GRANT ALL PRIVILEGES ON DATABASE book_db to peter;
template1=# GRANT ALL PRIVILEGES ON DATABASE document_db to peter;
template1=# \q
>> sudo -u peter psql -U peter -d book_db
book_db=>                                               NOTE : Prompt-marks for postgres and for peter are different.
book_db=> \q
>> sudo -u peter psql -U peter -d document_db
document_db=>
document_db=> \q
```

From my experience, only `postgres` can create database, while all users (including `peter`) can create table.

Part C. Create table, Insert and Select

After connecting a database, we can create table using following PostgreSQL command :

```
book_db=>
book_db=>CREATE TABLE   book_table (
             col_name0  col_type0(field_length0) col_constraint0,
             col_name1  col_type1(field_length1) col_constraint1,
             col_name2  col_type2(field_length2),
             col_name3  col_type3(field_length3));

For example
book_db=>CREATE TABLE   book_table (
             book_id    serial        PRIMARY KEY,
             book_name  varchar(20)   NOT NULL,
             catergory  varchar(20)   CHECK (catergory IN ('maths','programming','finance','economics','machine-learning')),
             buy_date   date);
```

- `SERIAL` is a PostgreSQL type, denoting an auto-incrementing integer
- `VARCHAR(20)` is a PostgreSQL type, denoting a fixed-length string
- `TEXT` is a PostgreSQL type, denoting a variable-length string
- `DATE` is a PostgreSQL type, denoting a date
- `TIMESTAMP` is a PostgreSQL type, denoting a datetime in `"yyyy-mm-dd hh:mm:ss.012345"`, in `sec/msec/usec/nsec` resolutions
- `PRIMARY KEY` is a PostgreSQL constraint, denoting a unique key for the row-data
- `NOT NULL` is a PostgreSQL constraint, denoting non-empty value
- `CHECK` is a PostgreSQL constraint, denoting one of the multiple values

```
book_db=> \d book_table
book_db=> INSERT INTO book_table (book_name, catergory, buy_date) VALUES ('stochastic calculus', 'maths',       '2012-01-01');
book_db=> INSERT INTO book_table (book_name, catergory, buy_date) VALUES ('complex analysis',    'maths',       '2013-01-01');
book_db=> INSERT INTO book_table (book_name, catergory, buy_date) VALUES ('C++ for finance',     'programming', '2014-01-01');
book_db=> SELECT * FROM book_table;

 book_id |      book_name      |  catergory  |  buy_date
---------+---------------------+-------------+-----------
     1   | stochastic calculus | maths       | 2012-01-01
     2   | complex analysis    | maths       | 2013-01-01
     3   | C++ for finance     | programming | 2014-01-01

book_db=> UPDATE book_table SET catergory = 'finance' WHERE book_id = 3;
book_db=> SELECT * FROM book_table;

 book_id |      book_name      |  catergory  |  buy_date
---------+---------------------+-------------+-----------
     1   | stochastic calculus | maths       | 2012-01-01
     2   | complex analysis    | maths       | 2013-01-01
     3   | C++ for finance     | finance     | 2014-01-01


book_db=> DELETE FROM book_table WHERE catergory = 'finance';
book_db=> SELECT * FROM book_table;

 book_id |      book_name      |  catergory  |  buy_date
---------+---------------------+-------------+-----------
     1   | stochastic calculus | maths       | 2012-01-01
     2   | complex analysis    | maths       | 2013-01-01


book_db=> ALTER TABLE book_table ADD price real;
book_db=> SELECT * FROM book_table;

 book_id |      book_name      |  catergory  |  buy_date  | price
---------+---------------------+-------------+------------+-------
     1   | stochastic calculus | maths       | 2012-01-01 |
     2   | complex analysis    | maths       | 2013-01-01 |
```

Please note that batch-insert is faster :

```
book_db=> INSERT INTO book_table (book_name, catergory, buy_date)
          VALUES ('stochastic calculus', 'maths',       '2012-01-01'),
                 ('complex analysis',    'maths',       '2013-01-01'),
                 ('C++ for finance',     'programming', '2014-01-01');
```

Three essential things in a table

For each table, there are 3 essential things :

- column name corresponds to `pod::m0 m1 m2`
- column type corresponds to `T0 T1 T2`
- value of a row data corresponds to `x0 x1 x2`

```
struct pod { T0 m0; T1 m1; T2 m2; };
std::array<pod,100> table;
table[n] = T{x0,x1,x2};
```

Part D. C++ API to PostgreSQL

According to most materials on the web, the C++ API can be installed by :

```
>> sudo apt install libpq-dev
```

However in my case, I cannot find `libpq.a` after installation :

```
>> find -uname libpq.a 2>/dev/null
nothing ...
```

Perhaps we need to build it from source code. Finally I chose to download from :

```
https://www.enterprisedb.com/download-postgresql-binaries
```

Unzip it, check header path and check library path :

```
>> tar -xvzf postgresql-10.15-1-linux-x64-binaries.tar.gz
>> cd pgsql
>> ls -al include                             // list all headers                such as libpq-fe.h
>> ls -al lib                                 // list all .a and .so             such as libpq.a
>> ls -al bin                                 // list all PostgreSQL utilities    such as createuser/createdb ...
```

Steps to use `libpq.a`

We can then write our C program to access database using `libpq.a`, but remember :

- `#include <libpq-fe.h>`   in .cpp
- `include_directories(~/dev/pgsql/include)`   in CMakeLists.txt as include path
- `target_link_libraries(my_exe -L~/dev/pgsql/lib)`   in CMakeLists.txt to library path of libpq.a
- `target_link_libraries(my_exe -lpq)`   in CMakeLists.txt to link libpq.a
- run your program by `sudo -u peter ./Test`   see the following remark

Suppose I am using linux account `dick` :

- write a C++ programme under `/home/dick/algo/build/debug/Test` to access PostgreSQL database
- while the connection string inside `/home/dick/algo/build/debug/Test` is ...

```
PGconn *conn = PQconnectdb("user=peter password=12qwasZX dbname=book_db");
```

- which means it connects to PostgreSQL server using PostgreSQL account `peter` (instead of PostgreSQL account `dick`)
- as this programme exists under linux account `dick` only, hence it should be run as :

```
~~>> ./Test~~                                     // linux account (ktchow1) ≠ PSQL account (peter), fail peer-authenication
>> sudo  -u peter ./Test                     // run ./Test under linux account ktchow1 using peter's role
~~>> sudo -iu peter ./Test~~                       // run ./Test under linux account peter, but file-not-found
```

Sample program

Remember to `PQclear(result)` if we reuse `result` for next `PQexec`.

```
    // Step 1 : Make Connection
    PGconn *connection = PQconnectdb("user=dick password=12qwasZX dbname=book_db");
    if (PQstatus(connection) == CONNECTION_BAD)
    {
        std::cout << "Connection to database failed : " << PQerrorMessage(connection);
        std::cout << "Don't forget to : ";
        std::cout << "1. start postgresql daemon";
        std::cout << "2. sudo -u username";
        PQfinish(connection);
        return;
    }
    else
    {
        std::cout << "Connection to database succeed : version " << PQserverVersion(connection);
        std::cout << "username : " << PQuser(connection);
        std::cout << "database : " << PQdb  (connection);
        std::cout << "password : " << PQpass(connection);
    }
```

```
// Step 2 : Drop table
PGresult *result = PQexec(connection, "DROP TABLE IF EXISTS book_table");
if (PQresultStatus(result) != PGRES_COMMAND_OK)
{
    PQclear(result);
    PQfinish(connection);
    return;
}
PQclear(result);



// Step 3 : Create table
std::string str;
str  = "CREATE TABLE book_table("s;
str += "book_id   SERIAL PRIMARY KEY, "s;
str += "book_name TEXT NOT NULL, "s;
str += "catergory VARCHAR(50) check (catergory in ('maths','programming','finance','machine learning')), "s;
str += "buy_time  TIMESTAMP)"s;

result = PQexec(connection, str.c_str());
if (PQresultStatus(result) != PGRES_COMMAND_OK)
{
    PQclear(result);
    PQfinish(connection);
    return;
}
PQclear(result);



// Step 4 : Insertion
std::vector<std::string> title     = ... please init here
std::vector<std::string> catergory = ... please init here (same size as above)

for(std::uint32_t n=0; n!=title.size(); ++n)
{
    str  = "INSERT INTO book_table (book_name, catergory, buy_time) VALUES (";
    str += "'"s += title[n] += "', "s;
    str += "'"s += catergory[n] += "', "s;
    str += "'"s += std::to_string(2010 + n) += "-01-01"s            // yyyy-mm-dd space
                += " 12:00:"s += std::to_string(10+n)               // hh:mm:ss
                += ".0000"s   += std::to_string(20+n) += "') "s;    // .usec

    result = PQexec(connection, str.c_str());
    if (PQresultStatus(result) != PGRES_COMMAND_OK)
    {
        std::cout << PQresultErrorMessage(result);
        PQclear(result);
        PQfinish(connection);
        return;
    }
    PQclear(result);
}



// Step 5 : Selection
result = PQexec(connection, "SELECT * FROM book_table WHERE buy_time > '2011-06-30 12:00:00.000000' ");
if (PQresultStatus(result) != PGRES_TUPLES_OK) // check against a different macro
{
    std::cout << PQresultErrorMessage(result);
    PQclear(result);
    PQfinish(connection);
    return;
}
else
{
    std::uint32_t NumRows = PQntuples(result);
    for(std::uint32_t n=0; n!=NumRows; ++n)
    {
        std::string str0 = PQgetvalue(result, n, 0);
        std::string str1 = PQgetvalue(result, n, 1);
        std::string str2 = PQgetvalue(result, n, 2);
        std::string str3 = PQgetvalue(result, n, 3);
    }
    PQclear(result);
}
```

```
    // Step 6 : Selection 1 row for META-DATA
    result = PQexec(connection, "SELECT * FROM book_table LIMIT 1"); // LIMIT means take first n rows only
    if (PQresultStatus(result) != PGRES_TUPLES_OK)
    {
        std::cout << PQresultErrorMessage(result);
        PQclear(result);
        PQfinish(connection);
        return;
    }
    else
    {
        std::uint32_t NumCols = PQnfields(result);
        for(std::uint32_t n=0; n!=NumCols; ++n)
        {
            std::string str = PQfname(result, n);
            std::cout << "\nCol_" << n << " is " << str;
        }
        PQclear(result);
    }



    // Step 7 : List tables in database
    result = PQexec(connection, "CREATE TABLE dummy_table0(id serial PRIMARY KEY, name VARCHAR(50) NOT NULL)");   PQclear(result);
    result = PQexec(connection, "CREATE TABLE dummy_table1(id serial PRIMARY KEY, name VARCHAR(50) NOT NULL)");   PQclear(result);
    result = PQexec(connection, "CREATE TABLE dummy_table2(id serial PRIMARY KEY, name VARCHAR(50) NOT NULL)");   PQclear(result);

    result = PQexec(connection, "SELECT table_name FROM information_schema.tables WHERE table_schema = 'public'");
    if (PQresultStatus(result) != PGRES_TUPLES_OK)
    {
        std::cout << PQresultErrorMessage(result);
        PQclear(result);
        PQfinish(connection);
        return;
    }
    else
    {
        std::uint32_t NumRows = PQntuples(result);
        for(std::uint32_t n=0; n!=NumRows; ++n)
        {
            std::cout << "\nTable_ " << n << " " << PQgetvalue(result, n, 0);
        }
        PQclear(result);
    }



    // Transaction is a sequence of SQL-statements grouped to form an ATOMIC operation.
    // Transaction is declared by :

/*  result = PQexec(conn, "BEGIN");
    transaction statement 0
    transaction statement 1
    ...
    transaction statement N
    result = PQexec(conn, "COMMIT"); */

    // Terminate
    PQfinish(connection);
```

Part E. Wrapped API to PostgreSQL

In order to avoid huge amount of code duplication in string processing to create various SQL statements, so for the sake of binding a POD `struct` to a PostgreSQL table, which facilitate both `INSERT` and `SELECT`, a template wrapper class is designed. This class helps to construct SQL statement in very user friendly way and also reduces the chance of having bugs in SQL statement, which can only be detected in SQL runtime. First of all we should think about how we are going to use it in client perspective.

```
// Step 1 for client - Define POD
enum class book_genre : std::uint32_t { maths, prog, quant };
struct book
{
    book_genre     genre;
    std::string    name;
    std::string    author;
    std::string    date_time;
    std::uint32_t  version;
};

// Step 2 for client - Connect DB
system("sudo service postgresql start");
PGconn* connection = PQconnectdb("user=dick password=12qwasZX dbname=book_db");
if (!psql::check_status(connection)) return;

// Step 3 for client - Construct API object. Define correspondence "key vs member pointer" with psql_item
psql::psql db remark1
{
    connection, "my_quant_library",
    psql::psql_item("genre",     &book::genre),          remark1
    psql::psql_item("name",      &book::name),           remark1
    psql::psql_item("author",    &book::author),         remark1
    psql::psql_item("date_time", &book::date_time),      remark1
    psql::psql_item("version",   &book::version)         remark1
};

// Step 4 for client - Create/Insert/Select as wish
std::vector<book> input;
std::vector<book> output;
input.emplace_back(book_genre::maths, "advanced calculus", "A.B.", "2012-06-01 12:30:00", 1);
input.emplace_back(book_genre::maths, "complex analysis",  "A.K.", "2013-07-01 13:40:10", 2);
input.emplace_back(book_genre::prog,  "design pattern",    "C.J.", "2014-08-01 14:50:20", 3);
input.emplace_back(book_genre::prog,  "c++ in a month",    "J.J.", "2015-09-01 15:00:30", 4);
input.emplace_back(book_genre::quant, "derivatives",       "T.O.", "2017-11-01 17:20:50", 5);

if (!db.create<book>())    { PQfinish(connection);  return; }
if (!db.insert(input))     { PQfinish(connection);  return; }
if (!db.select(output))    { PQfinish(connection);  return; }
for(const auto& x:output)    std::cout << x;
PQfinish(connection);
```

Secondly given above usage, we can confirm the interface as following header file, with a helper `struct` and 3 helper functions :

```
// There is 1 helper struct :
template<typename MPTR> struct psql_item requires (MPTR ptr) { *ptr; }
{
    using ptr_type = MPTR;
    std::string key;
    MPTR mptr;
};

// There are 3 helper functions :
// 1. convert type  into psql-string - for create table
// 2. convert value into psql-string - for insert
// 3. convert psql-string into value - for select
template<typename T> std::string psql_type()                { return "INVALID"; }
template<> inline    std::string psql_type<std::uint32_t>() { return "INTEGER"; }
template<> inline    std::string psql_type<std::uint64_t>() { return "INTEGER"; }
template<> inline    std::string psql_type<std::string>()   { return "TEXT";    }

template<typename T> std::string psql_value(const T& x)              { return "INVALID";         }
template<> inline    std::string psql_value(const std::uint32_t& x)  { return std::to_string(x); }
template<> inline    std::string psql_value(const std::uint64_t& x)  { return std::to_string(x); }
template<> inline    std::string psql_value(const std::string&   x)  { return x;                 }

template<typename T> void psql_fill_value(const std::string& s, T& x) {}
template<> inline    void psql_fill_value(const std::string& s, std::uint32_t& x) { x = std::stoul (s); }
template<> inline    void psql_fill_value(const std::string& s, std::uint64_t& x) { x = std::stoull(s); }
template<> inline    void psql_fill_value(const std::string& s, std::string&   x) { x = s; }
// Keyword "inline" is needed for template specialization in header file. Otherwise multiple-definition error.
```

One `psql_item` ceorresponds to one column in PostgreSQL table. Thus one table contains a tuple of `psql_item`, not a vector of `psql_item`, because in general, the columns in the table are of different types, thus homogenous vector cannot be used.

```
// Each ITEMS is one psql_item<MPTR>.
template<typename... ITEMS> class psql
{
public:
    using tuple_type = std::tuple<typename ITEMS::ptr_type ...>;
    psql(PGconn* connection_, const std::string& table_name_, const ITEMS&... items);

    // All 3 main functions are template members, all of them use std::apply inside, but in different ways.
    template<typename POD> bool create() const;
    template<typename POD> bool insert(const std::vector<POD>& entries) const;
    template<typename POD> bool select(std::vector<POD>& output) const;

private:
    mutable PGconn* connection;
    mutable PGresult* result;
    std::string table_name;

private:
    std::vector<std::string> keys; // This is a vector, as all elements are std::string.
    tuple_type mptrs;              // This is a tuple,  as all elements are different types.
};
```

Finally we can implement the 3 main functions `create/insert/select`. The implementation involves the following C++ tricks :

- template `POD` construction using `CTAD` without specifying the template parameters see `remark1`
- template `POD` member access using member pointer see `remark2`
- template `POD` member type extraction using a combo of `decltype` and `declval` see `remark3`
- conversion of single template type `std::tuple<ARGS...>` into variadic parameter pack `ARGS...` using `std::apply` see `remark4`
- with above conversion, we can then apply ***fold expression with comma operator*** on the `ARGS...` see `remark5`

Here is the constructor, which separates the variadic arguments into a vector of `keys` and a tuple of member pointers `mptrs`. Variadic syntax works fine, no `std::apply` is needed.

```
template<typename... ITEMS>
psql::psql<ITEMS...>::psql(PGconn* conn, const std::string& table, const ITEMS&... items)
: connection(conn), table_name(table)
{
    (keys.push_back(items.key),...); remark5
    mptrs = std::make_tuple(items.mptr ...);
}
```

Implementation of `create/insert/select` are pretty similar, the main difference is the content inside ***fold expression***. Code for `create` is shown below, refer to my `git` repository for implementation of `insert` and `select`, both of which involve extra loop for multi-rows.

```
template<typename... ITEMS>
template<typename POD>
bool psql::psql<ITEMS...>::create() const
{
    // *** Part 1 *** //
    std::stringstream ss;
    std::uint32_t n=0;
    std::apply
    (
        [this, &ss, &n](const typename ITEMS::ptr_type&... unpacked_mptrs)
        {
            ss << "(";
            ((
                ss << keys[n++] << " "
                   << psql_type<std::remove_cvref_t
                               <decltype(std::declval<POD>().*unpacked_mptrs)>>() remark2 and remark3
                // << psql_type<decltype(std::declval<POD>().*unpacked_mptrs)>()  // Does not work, return INVALID
                   << (n!=keys.size()? ", " : "")
            ),...); remark5
            ss << ")";
        }, mptrs
    ); remark4

    // *** Part 2 *** //
    using namespace std::string_literals; // for string literal
    std::string   drop_str("DROP TABLE IF EXISTS");  drop_str += " "s += table_name;
    std::string create_str("CREATE TABLE");        create_str += " "s += table_name += " "s += ss.str();

    // *** Part 3 *** //
    result = PQexec(connection, drop_str.c_str());
    if (!check_status("Drop table", connection, result, PGRES_COMMAND_OK)) return false;
    PQclear(result);

    result = PQexec(connection, create_str.c_str());
    if (!check_status("Create table", connection, result, PGRES_COMMAND_OK)) return false;
    PQclear(result);
    return true;
}
```

Part F. Join Tables

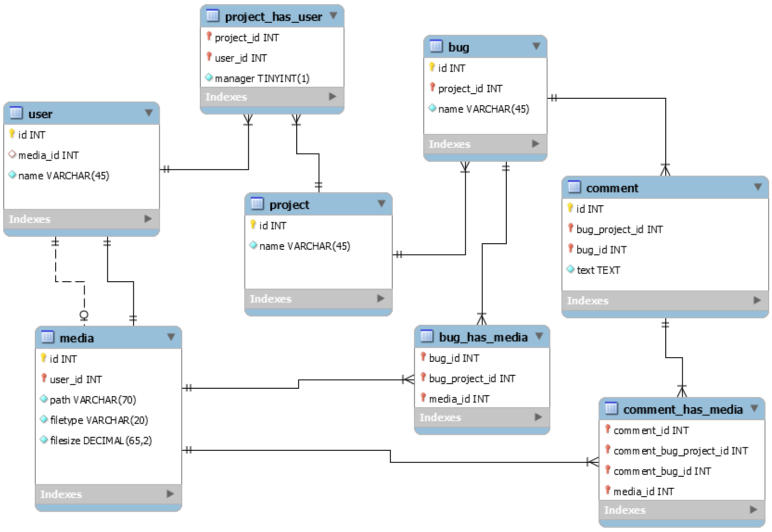
A relational database is decomposed into multiple tables, for the sake of simplicity, avoid duplicated data and ensure single source of truth. The tables are related and govered by a high level schema.

```
 employee | id | salary | join_date | team
----------+----+--------+-----------+------
   Dave   | 01 | 10000  | 20020101  | ABC  ---+
   Mary   | 02 | 20000  | 20030401  | DEF  ---|---+
   Paul   | 03 | 40000  | 20060701  | ABC  ---+   |
   Jack   | 04 | 25000  | 20130415  | DEF     |   |
                                              |   |
   team   | tid | project | budget            |   | schema governing relationship
----------+-----+---------+--------           |   |
   ABC    | 01  | trading | 100M           <--+   |
   DEF    | 02  | quant   | 200M           <------+
```

Instead of all-in-one table, which couples everything and duplicates some data :

```
 employee | id | salary | join_date | team | tid | project | budget
----------+----+--------+-----------+------+-----+---------+--------
   Dave   | 01 | 10000  | 20020101  | ABC  | 01  | trading | 100M
   Mary   | 02 | 20000  | 20030401  | DEF  | 02  | quant   | 200M
   Paul   | 03 | 40000  | 20060701  | ABC  | 01  | trading | 100M
   Jack   | 04 | 25000  | 20130415  | DEF  | 02  | quant   | 200M
```

For bigger database, schema will look like the following :



As a database is segregated into multiple tables, `JOIN` table is therefore a necessary feature. There are 5 different `JOIN` :

- cross join = product of `tableA` and `tableB`
- inner join = sum of `tableA` and `tableB` (intersection set, i.e. logical *AND*)
- left outer join = sum of `tableA` and `tableB` (`tableA` plus intersection set)
- right outer join = sum of `tableA` and `tableB` (`tableB` plus intersection set)
- full outer join = sum of `tableA` and `tableB` (union set, i.e. logical *OR*)

```
str0  = "INSERT INTO item (order_id, prod_type) VALUES";
str0 += "('ABC_0001', 'PROD_A' ),"s;
str0 += "('ABC_0001', 'PROD_B' ),"s;
str0 += "('ABC_0001', 'PROD_B' ),"s;
str0 += "('ABC_0002', 'PROD_A' ),"s;
str0 += "('ABC_0002', 'PROD_C' ),"s;
str0 += "('ABC_0003', 'PROD_B' ),"s;
str0 += "('ABC_0003', 'UNKNOWN'),"s;
str0 += "('ABC_0004', 'PROD_C' );"s;

str1  = "INSERT INTO product (prod_type, price) VALUES";
str1 += "('PROD_A',  12.50),"s;
str1 += "('PROD_B',  19.99),"s;
str1 += "('PROD_C',  25.00),"s;
str1 += "('PROD_D',  85.00),"s;
str1 += "('PROD_E', 199.99);"s;

result0 = PQexec(connection, str0.c_str());
result1 = PQexec(connection, str1.c_str());
```

```
for(std::uint32_t n=0; n!=5; ++n)
{
    std::string s;
    if (n==0) s = "SELECT * FROM item       CROSS JOIN product";
    if (n==1) s = "SELECT * FROM item       INNER JOIN product ON item.prod_type = product.prod_type ORDER BY item.item_id";
    if (n==2) s = "SELECT * FROM item LEFT  OUTER JOIN product ON item.prod_type = product.prod_type ORDER BY item.item_id";
    if (n==3) s = "SELECT * FROM item RIGHT OUTER JOIN product ON item.prod_type = product.prod_type ORDER BY item.item_id";
    if (n==4) s = "SELECT * FROM item FULL  OUTER JOIN product ON item.prod_type = product.prod_type ORDER BY item.item_id";

    PGresult* result = PQexec(connection, s.c_str());
    for(std::uint32_t n=0; n!=PQntuples(result); ++n)
    {
        std::cout << PQgetvalue(result, n, 0) << PQgetvalue(result, n, 1) << ... << PQgetvalue(result, n, 5);
    }
    PQclear(result);
}
```

Here are the results of various join.

```
// Cross join (all combinations)
1  ABC_0001  PROD_A   1  PROD_A  12.50
2  ABC_0001  PROD_B   1  PROD_A  12.50
3  ABC_0001  PROD_B   1  PROD_A  12.50
4  ABC_0002  PROD_A   1  PROD_A  12.50
5  ABC_0002  PROD_C   1  PROD_A  12.50
6  ABC_0003  PROD_B   1  PROD_A  12.50
7  ABC_0003  UNKNOWN  1  PROD_A  12.50
8  ABC_0004  PROD_C   1  PROD_A  12.50
1  ABC_0001  PROD_A   2  PROD_B  19.99
2  ABC_0001  PROD_B   2  PROD_B  19.99
3  ABC_0001  PROD_B   2  PROD_B  19.99
...
6  ABC_0003  PROD_B   5  PROD_E 199.99
7  ABC_0003  UNKNOWN  5  PROD_E 199.99
8  ABC_0004  PROD_C   5  PROD_E 199.99 number of rows = 8*5 = 40

// Inner join
1  ABC_0001  PROD_A   1  PROD_A  12.50
2  ABC_0001  PROD_B   2  PROD_B  19.99
3  ABC_0001  PROD_B   2  PROD_B  19.99
4  ABC_0002  PROD_A   1  PROD_A  12.50
5  ABC_0002  PROD_C   3  PROD_C  25.00
6  ABC_0003  PROD_B   2  PROD_B  19.99
8  ABC_0004  PROD_C   3  PROD_C  25.00 number of rows < 8+5

// Left Outer join (1 extra line as compared in inner join)
1  ABC_0001  PROD_A   1  PROD_A  12.50
2  ABC_0001  PROD_B   2  PROD_B  19.99
3  ABC_0001  PROD_B   2  PROD_B  19.99
4  ABC_0002  PROD_A   1  PROD_A  12.50
5  ABC_0002  PROD_C   3  PROD_C  25.00
6  ABC_0003  PROD_B   2  PROD_B  19.99
7  ABC_0003  UNKNOWN
8  ABC_0004  PROD_C   3  PROD_C  25.00 number of rows < 8+5

// Right Outer join (2 extra lines as compared in inner join)
1  ABC_0001  PROD_A   1  PROD_A  12.50
2  ABC_0001  PROD_B   2  PROD_B  19.99
3  ABC_0001  PROD_B   2  PROD_B  19.99
4  ABC_0002  PROD_A   1  PROD_A  12.50
5  ABC_0002  PROD_C   3  PROD_C  25.00
6  ABC_0003  PROD_B   2  PROD_B  19.99
8  ABC_0004  PROD_C   3  PROD_C  25.00
                      4  PROD_D  85.00
                      5  PROD_E 199.99 number of rows < 8+5

// Full Outer join (3 extra lines as compared in inner join)
1  ABC_0001  PROD_A   1  PROD_A  12.50
2  ABC_0001  PROD_B   2  PROD_B  19.99
3  ABC_0001  PROD_B   2  PROD_B  19.99
4  ABC_0002  PROD_A   1  PROD_A  12.50
5  ABC_0002  PROD_C   3  PROD_C  25.00
6  ABC_0003  PROD_B   2  PROD_B  19.99
7  ABC_0003  UNKNOWN
8  ABC_0004  PROD_C   3  PROD_C  25.00
                      4  PROD_D  85.00
                      5  PROD_E 199.99 number of rows < 8+5
```

Reference

Google : How To Install and Use PostgreSQL on Ubuntu 18.04, DigialOcean.

Part G. SQLite3
SQLite3 is a light weighed database hosted with a local file. Please use version 3 or latter. Lets start from beginning :

```
>> sudo apt install sqlite3
>> sqlite3                            start without database
>> sqlite3 my_test.db                 start with database
```

Open database in GUI :

```
>> sudo apt install sqlitebrowser
>> sqlitebrowser
```

Once SQLite3 is started, it prompts with `sqlite>`, we either enter a SQLite3 command or a SQL statement.

- SQLite3 command must start with a dot, also known as dot-command, which does adminstrative task
- SQL statement must end with a semicolon, otherwise it keeps prompting with `...>`, it is used for multi-line SQL statement

Here are some common dot commands :

```
sqlite> .open my_test.db              create a new database / open an existing database
sqlite> .database                     list all databases
sqlite> .table                        list all tables
sqlite> .schema                       list all schema, i.e. list all create-table statements for existing tables
sqlite> .mode list                    set output format as list, like :         peter|101|projectA
                                                                                david|202|projectB
                                                                                susan|303|projectC
sqlite> .mode quote                   set output format as quote, like :        peter, 101, projectA
                                                                                david, 202, projectB
                                                                                susan, 303, projectC
sqlite> .mode column                  set output format as column, like :       name   id   project
                                                                                ---------------------
                                                                                peter  101  projectA
                                                                                david  202  projectB
                                                                                susan  303  projectC
sqlite> .mode markdown                set output format as markdown, like :     postgres output
sqlite> .exit
```

Here are some common SQL statements :

```
sqlite> create table my_table (name varchar(10), id smallint, project varchar(10));
sqlite> insert into my_table values('peter', 101, 'projectA');
sqlite> insert into my_table values('david', 202, 'projectB');
sqlite> insert into my_table values('susan', 303, 'projectC');
sqlite> select * from my_table;
```

In SQLite3, there is a meta table known as `sqlite_schema` which stores the information about all tables, so dot commands are actually querys to the meta table. For example :

```
sqlite> .table
is equivalent to ...

sqlite> select name from sqlite_schema where
        type in ('table','view') and
        name not like 'sqlite_%'
        order by 1

sqlite> .schema
is equivalent to ...

sqlite> select sql from sqlite_schema
        order by tbl_name, type DESC, name
```

Please read `sqlite.org/cli.html` for details of SQLite3.

Part H. Using YAML cpp library and XML cpp library

Comparison of SQL table, C++ struct, csv, xml, yaml and json

Consider we have a SQL table, have 3 elements `col-name/col-type/row-value`, which correspond to `std::vector<pod>` in C++ as :

```
struct pod                                            std::vector<pod> table;
{//  2. SQL col-type  1. SQL col-name                 table.emplace_back(x00,x01,x02,x03,x04);
     std::uint32_t    entry_id;                       table.emplace_back(x10,x11,x12,x13,x14);
     std::uint32_t    group_id;                       table.emplace_back(x20,x21,x22,x23,x24); // 3. SQL row-values
     std::string      name;                           ...
     std::string      attr;                           3 ESSENTIAL things for SQL / POD / std::map
     std::uint32_t    value;
};
```

The 3 elements do also correspond to a `std::map<std::string,V>` in C++.

```
1. SQL col-name    =   key   in std::map<std::string,V>
2. SQL col-type    =   V     in std::map<std::string,V> i.e. typeof(value)=V, where typeof(key)=K=std::string
3. SQL row-values  =   value in std::map<std::string,V>
```

This is `csv`, readable by *Excel*.

```
entry_id,group_id,name,attr,value
1,101,peter,male,30
2,102,paul,male,24
3,103,mary,female,38
```

As a convention, we use :

- `yaml` for system input, i.e. config
- `json` for system output, i.e. `event POD` logging

They are different in nature :

- `yaml` supports comment (`json` not) which is a must for config
- `json` is the native representation of objects, thus good for logging

Open source cplusplus library for read/write of these files :

- `csv` with `d99kris / rapidcsv`
- `xml` with `discord / rapidxml`
- `yaml` with `jbeder / yaml-cpp`
- `json` with `nlohman / json`

All above are human-readable. Binary serialization includes :

- `cbor`
- `google protocol buffer` (I use this thing in `TDMS`)
- `message-pack` (I use this thing when installing `deoplete` for `nvim`)

This is `xml`, with nested tags.

```
<?xml version=”1.0”encoding=”utf-8”?>
<list>
  <entry>
      <entry_id> 1         </entry_id>
      <group_id> 101       </group_id>
      <name>     peter     </name>
      <attr>     male      </attr>
      <value>    30        </value>
  </entry>
  <entry>
      <entry_id> 2         </entry_id>
      <group_id> 102       </group_id>
      <name>     paul      </name>
      <attr>     male      </attr>
      <value>    24        </value>
  </entry>
  <entry>
      <entry_id> 3         </entry_id>
      <group_id> 103       </group_id>
      <name>     mary      </name>
      <attr>     female    </attr>
      <value>    38        </value>
  </entry>
</list>
```

This is `yaml`, which is like simplified `xml`, with tags replaced by indentation.

```
list:
- entry:
      entry_id: 1
      group_id: 101
      name:     peter
      attr:     male
      value:    30
- entry
      entry_id: 2
      group_id: 102
      name:     paul
      attr:     male
      value:    24
- entry
      entry_id: 3
      group_id: 103
      name:     mary
      attr:     female
      value:    38
```

This is `json`, which looks like SQL.

```
[
    { "entry_id": 1, "group_id": 101, "name": peter, "attr": male,   "value": 30 },
    { "entry_id": 2, "group_id": 102, "name": paul,  "attr": male,   "value": 24 },
    { "entry_id": 3, "group_id": 103, "name": mary,  "attr": female, "value": 38 }
]
```

Download and build YAML cpp library

First of all, `git` clone YAML cplusplus library and `cmake` it.

```
>> cd dev
>> git clone https://github.com/jbeder/yaml-cpp.git
>> cd yaml-cpp
>> mkdir build; cd build
>> mkdir debug; cd debug
>> cmake -DCMAKE_BUILD_TYPE=Debug ../..
```

2021 Mar 01

Problems I came across when using `yaml-cpp` for HKEX `omni-api` :

- `yaml-cpp` does not support `~/dev` folder (use `/home...` instead)
- `yaml.h` should be included before HKEX header `hkats.h`, reversing the order results in crash (for unknown reason)
- `config.yaml` with `'\t'` results in crash (hard to identify)

We can then find header and static library in the following folders :

```
    dev/yaml-cpp/include/yaml-cpp/yaml.h
    dev/yaml-cpp/build/debug/libyaml-cppd.a
```

We update the `CMakeLists.txt` as :

```
    include_directories(~/dev/yaml-cpp/include)
    target_link_libraries(Test "/home/ktchow1/dev/yaml-cpp/build/debug/libyaml-cppd.a")
#   target_link_libraries(Test             "~/dev/yaml-cpp/build/debug/libyaml-cppd.a") # This line does not work, why?
```

We include header in related `cpp` files as :

```
    #include <yaml-cpp/yaml.h>
```

Using YAML class

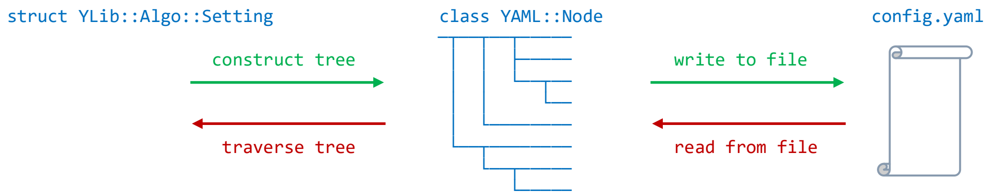
The main YAML class is called `YAML::Node`, it is a tree-traversor, which traverses through whole YAML tree. YAML takes a recursive form, which means each node in the YAML tree is also a subtree. There are 3 types for each node :

- `YAML::NodeType::Scalar` which is a scalar
- `YAML::NodeType::Sequence` which is a vector of scalar or a vector of `NodeType::Map`
- `YAML::NodeType::Map` which is a subtree

Therefore :

- `YAML::NodeType::Scalar` is a leaf
- `YAML::NodeType::Sequence` is a non leaf
- `YAML::NodeType::Map` is a non leaf

What we do with YAML are :



Lets go through the forward direction. Integer and string can be used interchangably, as everything will be stored as string in file.

- `YAML::Node::operator[]` returns `YAML::Node`
- scalar value can be a reference to another items in `yaml`, source is marked with `&n`, destination is marked with `*n`
- scalar value cannot be overwritten as sequence, it will crash
- sequence value can be overwritten as map, keys for existing items become `int` starting from zero *(see example below)*

```
    YAML::Node node;

    // (1) scalar
    node[ 100 ]    = 10000;                      // Both key and value can be int or string (interchangably).
    node["101"]    = 10000;
    node["101"]    = "abc";                      // Value can be overwritten.
    node["scalar"] = "abc";
//  node["scalar"].push_back(10000);             // Value cannot be overwritten as sequence, it crashes.

    // (2) sequence (i.e. heterogenous vector)
    node["seq"].push_back(10);
    node["seq"].push_back(11);
    node["seq"].push_back(12);
    node["seq"].push_back("abc");
    node["seq"].push_back("ABC");
    node["seq"].push_back(node[101]);            // Value can be a reference to another item in yaml. Indicated as & * in yaml.
```

```
// (3a) map (i.e. heterogenous subtree), method1 : construct directly from temporary values
node["map"][ 123 ] = 12345;                // node["map"] is a subtree
node["map"]["abc"] = "abcdef";
node["map"]["seq"].push_back(10);
node["map"]["seq"].push_back(11);
node["map"]["seq"].push_back(12);
node["map"]["map"]["key0"] = "value0";     // node["map"] is different from node["map"]["map"]
node["map"]["map"]["key1"] = "value1";
node["map"]["map"]["key2"] = "value2";
node["map"]["map"]["key3"] = "value3";
node["map"]["map"]["key4"] = node[101];    // reference to another node is created

// (3b) map (i.e. heterogenous subtree), method2 : construct from existing nodes n0-n4
YAML::Node n0,n1,n2,n3,n4;
n0["stock"] = "HSI";  n0["price"] = 10.123;  n0["other"] = "*****";
n1["stock"] = "HKA";  n1["price"] = 20.234;  n1["other"] = ".....";
n2["stock"] = "HKB";  n2["price"] = 30.345;  n2["other"] = "/////";
n3["stock"] = "HKC";  n3["price"] = 40.456;  n3["other"] = "-----";

node["subtree"].push_back(n0);             // node["subtree"] is a sequence
node["subtree"].push_back(n1);
node["subtree"].push_back(n2);
node["subtree"]["extra0"] = n3;            // node["subtree"] becomes a map, what happens to previous items?
node["subtree"]["extra1"] = node[101];     // node["subtree"]["0"]      = n0
                                           // node["subtree"]["1"]      = n1
// Write yaml-tree to yaml-file            // node["subtree"]["2"]      = n2
std::ofstream ofs("config.yaml");          // node["subtree"]["extra0"] = n3
ofs << node;
```

Lets go through the backward direction.

```
template<typename T> void access(const YAML::Node& node, const T& key)
{
    if (!node[key])
    {
        std::cout << "node[" << key << "] is null";
    }
    else if (node[key].Type() == YAML::NodeType::Scalar)
    {
        std::cout << "node[" << key << "] is scalar : " << node[key];
    }
    else if (node[key].Type() == YAML::NodeType::Sequence)
    {
        std::cout << "node[" << key << "] is sequence : ";
        for(std::uint32_t n=0; n!=node[key].size(); ++n) std::cout << node[key][n] << " ";
    }
    else if (node[key].Type() == YAML::NodeType::Map)
    {
        std::cout << "node[" << key << "] is map : " << node[key];
    }
}

YAML::Node node = YAML::LoadFile("config.yaml");
access(node, 100);
access(node, 101);                        // it can be accessed with integer-key
access(node, "scalar");
access(node, "seq");
access(node, "map");
access(node, "subtree");
```

Apart from printing, we can parse it into custom `struct` by template member function `YAML::Node::as<T>()`, `decltype` is useful here.

```
struct element
{
    std::uint32_t n;
    std::string s;
    std::vector<std::uint32_t> v;
    std::map<std::string, std::string> m;
};

void parse_into(const YAML::Node& node, element& x)
{
    x.n = node[ 123 ].as<decltype(element::n)>();
    x.s = node["abc"].as<decltype(element::s)>();
    for(std::uint32_t n=0; n!=node["seq"].size(); ++n)
    {
        x.v.push_back(node["seq"][n].as<typename decltype(element::v)::value_type>());
    }
    x.m["key0"] = node["map"]["key0"].as<typename decltype(element::m)::mapped_type>();
    x.m["key1"] = node["map"]["key1"].as<typename decltype(element::m)::mapped_type>();
    x.m["key2"] = node["map"]["key2"].as<typename decltype(element::m)::mapped_type>();
    x.m["key3"] = node["map"]["key3"].as<typename decltype(element::m)::mapped_type>();
    x.m["key4"] = node["map"]["key4"].as<typename decltype(element::m)::mapped_type>();
}

element x;
parse_into(node["map"], x);
```

Download and build XML cpp library

There are two popular XML library

- `libxml2` used in Daiwa
- `rapidxml` used in Yubo

Lets take a look at the first one. Download from `http://xmlsoft.org/download` to folder `download`.

```
>> cd ~/temp
>> mv download/libxml2-2.7.1.tar.gz .
>> tar -xvzf libxml2-2.7.1.tar.gz
>> cd libxml2-2.7.1
>> ./configure --prefix=/usr/local/libxml2
>> make [This is the main step building libxml2, there are a lot of warnings, but its ok.]
>> sudo make install
```

where `/usr/local/libxml2` is the location for placing the `libxml2` library. After that inside `CMakeLists.txt`, we

1. add include path
2. add library path with flag `-L`
3. link library `libxml2` with flag `-l` (for both .a and .so)

```
target_include_directories(Test PUBLIC
    /usr/local/libxml2/include/libxml2
)

target_link_libraries(Test -L/usr/local/libxml2/lib)
target_link_libraries(Test -lxml2)
```

Using XML class

This is a C style library. Its better to wrap it up into classes.

Part I. Using Reckless log library

What is Reckless log?

Reckless log is a low latency logger, implemented with lockfree disruptor under the hood. It divides items into :

- header *which will be printed in every line, specified as template parameter when we instantiate logger*
- argument *which will be printed on demand, through one formatting string plus variadic arguments*

Reckless log decouples the process into :

- logger *which format the final string depending on formatting string and arguments*
- writer *which write the final string into files, socket etc (file descriptor)*

Both tasks are slow, both are done by a separate thread, called the reckless thread, spawned by reckless logger.

- when we call logging using our app thread, it simply pushes the formatting string together with arguments into a `mpscq`
- when reckless thread does the formatting and writing, it pops from the `mpscq`

Both header and argument can be customized :

- we can create our own `struct` for header / argument and tell reckless how to format it
- the formatting function for custom header / custom argument are slightly different

Normal usage

Here is a normal usage. Firstly instantiate a writer with filename, then instantiate a logger with the writer.

```
// *** Step 1 - Instantiate a file_writer *** //
reckless::file_writer writer("reckless.log");

// *** Step 2 - Instantiate a severity_log *** //
reckless::severity_log
<
    reckless::indent<4U>,       // indentation
    ' ',                        // delimitor
    reckless::timestamp_field,  // use timestamp as 1st header
    reckless::severity_field    // use severity  as 2nd header
>
logger(&writer);

logger.debug("price model m=%f c=%f", 0.123, 0.456);
{
    reckless::scoped_indent indent;
    logger.info ("hitter place price=%d quant=%d", 100, 200);
    logger.info ("hitter place price=%d quant=%d", 101, 201);
    {
        reckless::scoped_indent indent;
        logger.info ("hitter place price=%d quant=%d", 102, 202);
    }
}
logger.warn("fitter %s", "unused 1");
logger.warn("fitter %s", "unused 2");
using namespace std::string_literals;
logger.error("quoter %s", "runtime_error_"s + std::to_string(301));
logger.error("quoter %s", "runtime_error_"s + std::to_string(302));
```

Here is the output log.

```
2021-02-04 20:33:04.077 D price model m=0.123 c=0.456
2021-02-04 20:33:04.390 I     hitter place price=100 quant=200
2021-02-04 20:33:04.390 I     hitter place price=101 quant=201
2021-02-04 20:33:04.390 I         hitter place price=102 quant=202
2021-02-04 20:33:04.691 W fitter unused 1
2021-02-04 20:33:04.691 W fitter unused 2
2021-02-04 20:33:04.991 E quoter runtime_error_301
2021-02-04 20:33:04.991 E quoter runtime_error_302
```

Customization of argument

The customization of argument is easier. What we need to do is to define our `struct`, then provide global function. Multiple formats is supported. As shown in this example, we define `%a`, `%b`, `%c` and `%d` for `struct rgb`. There is an example in its Git repo.

```
struct rgb { std::uint8_t r; std::uint8_t g; std::uint8_t b; };

const char* format(reckless::output_buffer* buffer, const char* formatting_str, const rgb& x)
{
    if      (*formatting_str == 'a') { reckless::template_formatter::format ( buffer, "(%d,%d,%d)", x.r, x.g, x.b ); }
    else if (*formatting_str == 'b') { reckless::template_formatter::format ( buffer, "(%d|%d|%d)", x.r, x.g, x.b ); }
    else if (*formatting_str == 'c') { reckless::template_formatter::format ( buffer, "[%d|%d|%d]", x.r, x.g, x.b ); }
    else if (*formatting_str == 'd') { reckless::template_formatter::format ( buffer, "<%d-%d-%d>", x.r, x.g, x.b ); }

    return formatting_str + 1; // This is a must.
}

custom_arg x(120,180,240);
logger.debug("x is %a", x);    // 2021-02-04 20:33:04.991 D x is (120,180,240)
logger.debug("x is %b", x);    // 2021-02-04 20:33:04.991 D x is (120|180|240)
logger.debug("x is %c", x);    // 2021-02-04 20:33:04.991 D x is [120|180|240]
logger.debug("x is %d", x);    // 2021-02-04 20:33:04.991 D x is <120-180-240>
```

Customization of header

The customization of header is more difficult, there is no tutorial. However, Alu somehow hacked it as the following.

```
template<std::uint32_t M, std::integral T> // M = number of digits, N = number to be written on buffer
void int_to_ascii(char* pc, T N)
{
    for(std::uint32_t m=0; m!=M; ++m)
    {
        pc[M-1-m] = '0' + N%10;
        N = N/10;
    }
}

struct rgb_header
{
    // This format function is what reckless logger needs.
    inline bool format(reckless::output_buffer* buffer)
    {
        char* pc = buffer->reserve(M*3+4);
        pc[0]     = '[';   int_to_ascii<M>(pc    +1, x);
        pc[M+1]   = '-';   int_to_ascii<M>(pc+M  +2, y);
        pc[M*2+2] = '-';   int_to_ascii<M>(pc+M*2+3, z);
        pc[M*3+3] = ']';   buffer->commit(M*3+4);
        return true;
    }
    static const std::uint32_t M = 3;
    std::uint8_t x;
    std::uint8_t y;
    std::uint8_t z;
};

struct xyz_header
{
     // similarly we can define another header
};
```

After that we have to define our logger class, which is derived from `reckless::basic_log`. What we need is to define a custom logging function to forward logging data to the `write` function of base class `reckless::basic_log`.

```
class custom_log : public reckless::basic_log
{
public:
    custom_log(reckless::file_writer* reckless_writer) : reckless::basic_log(reckless_writer) {}

    // Forward the input of my custom log function to base class's write function
    template<typename... ARGS> void my_custom_log(rgb_header0&& rgb,
                                                  xyz_header0&& xyz,
                                                  const char* formatting_str, ARGS&&... args)
    {
        reckless::basic_log::write<reckless::policy_formatter<reckless::indent<4U>, ' ', rgb_header, xyz_header>>
        (
            std::move(rgb),
            std::move(xyz),
            reckless::indent<4U>(), // We don't know why this is needed, just copy from reckless internal.
            formatting_str,
            std::forward<ARGS>(args)...
        );
    }
};
```

There is no need to modify writer. This is how we use custom_log.

```
// Step 1 - Instantiate a file_writer
reckless::file_writer writer("reckless2.log");

// Step 2 - Instantiate a custom_log
custom_log logger(&writer);

logger.my_custom_log(rgb_header(r0,g0,b0), xyz_header(x0,y0,z0), "name=%s value=%d", str0, i0);
logger.my_custom_log(rgb_header(r1,g1,b1), xyz_header(x1,y1,z1), "step=%s state=%d model=%f", str1, state1, model1);
logger.my_custom_log(rgb_header(r2,g2,b2), xyz_header(x2,y2,z2), "step=%s state=%d model=%f", str2, state2, model2);
```

Problem with string

There is a reported bug for reckless log to log std::string, probably due to the SSO in the string. We try work around by introducing a thin string wrapper as the following. Besides, we make it non copyable so as to force it to move (faster). There are two approaches, either implemented with raw pointer (you then need to handle all move semantics and delete), or with unique pointer (everything is automatic). Beware it is a unique pointer of array, a new feature in C++17.

```
struct custom_str
{
    custom_str() = delete;
   ~custom_str()
    {
        if (ptr!=nullptr) delete [] ptr;
    }

    custom_str(const custom_str&) = delete;
    custom_str(custom_str&& rhs) : size(rhs.size), ptr(rhs.ptr)
    {
        rhs.size = 0;
        rhs.ptr  = nullptr;
    }

    custom_str& operator=(const custom_str&) = delete;
    custom_str& operator=(custom_str&& rhs)
    {
        std::swap(size, rhs.size);
        std::swap(ptr, rhs.ptr);
        return *this;
    }

    explicit custom_str(const std::string& str) : size(str.size()), ptr(new char[size+1])
    {
        memcpy(ptr, str.c_str(), size);
        ptr[size] = '\0';
    }

    std::uint32_t size;
    char* ptr;
};

struct custom_str2 // simpler implementation than previous one
{
    custom_str2() = delete;
   ~custom_str2() = default;
    custom_str2(const custom_str2&) = delete;
    custom_str2(custom_str2&&) = default;
    custom_str2& operator=(const custom_str2&) = delete;
    custom_str2& operator=(custom_str2&&) = default;

    explicit custom_str2(const std::string& str) : size(str.size()), uptr(new char[size+1])
    {
        memcpy(uptr.get(), str.c_str(), size);
        uptr.get()[size] = '\0';
    }

    std::uint32_t size;
    std::unique_ptr<char[]> uptr;
};
```