c7json

*C++ JSON*エンコード&デコード機能

概要

特徴

c7json では、C++のデータ構造を JSON形式との間で交換可能にするためのクラス群、そのクラスオブジェクトとファイルや iostream との間でデータ交換を行う関数群、それらクラス群を簡易言語から生成するスクリプトを提供する。

- 目的は C++ の 整数、浮動小数点数、文字列、構造体、unordered_map, vector, tuple, pair, 時刻値、バイト列から JSON形式へのマッピングであり、任意の JSON形式を扱うことは考えていない。
- 型を強制することで、安全性とプログラミングのやり易さを確保する。
- 構造体、unordered_map は JSONのオブジェクトとして交換する。
- vector, tuple, pair は JSONの配列として交換する。
- 文字列(UTF-8)は JSONエンコード規則に従った文字列として交換する。
- 時刻値は libc7++の時刻(マイクロ秒単位の64bit整数)とし、これを ISO8601形式で変換する。
- バイト列(バイナリデータ)は std::vector<uint8_t> とし、これを BASE64エンコーデッド文字列で変換する。
- C++構造体として不明なメンバー名(JSON上のキー)を許容する型と、エラーとする型の二種を用意する。前者は、一部だけに関心がある場合に、その部分だけの定義で利用することが可能で、不明なメンバーのデータも書き込み時に正確にされる。後者はそのような機能を持たない分、フットプリントを小さくできる。
- 専用の DSN(Data Structure Notation) という簡易DSL(Domain Specific Language)で C++のブロックコメントに埋め込んだデータ構造の定義から、クラス定義を生成することが可能。

具体例

例として、音楽ライブラリのデータをJSONで保存することを考えてみる。C++の型定義を先に考え、それを c7json による JSON エンコード/デコード用の型で再定義することで、使用感を掴んでもらおう。

C++ で定義するなら…

Song 型は、楽曲ID, 曲名、演奏時間(秒)、音楽データ、お気に入りの5つを保持する。

```
using SongID = int;

struct Song {
    SongID                id;             // 楽曲ID
    std::string           title;          // 楽曲名
    int                   duration_s;     // 演奏時間
    bool                  favorite;       // お気に入り
    std::vector<uint8_t>  audio;          // 音楽データ
};
```

Album 型は、アルバムID、アルバムタイトル、アーティスト名、リリース日、楽曲IDの列を保持する。

```
using AlbumID = int;

struct Album {
    AlbumID             id;             // アルバムID
    std::string         title;          // アルバムタイトル
    std::string         artist;         // アーティスト名
    c7::usec_t          release;        // リリース日
    std::vector<SongID> songs;          // 全楽曲
};
```

Library 型はIDから音楽やアルバム情報への辞書のほか、運用時の情報を保持する。history が vector でなく unordered_map だったり、map にいたっては座標と楽曲の関係など意味不明だと思うが、あくまでテストを目的とする作為的なものなので意味は考えないこと。

```
using Location = std::tuple<int, int, int>;           // x, y, z

struct Library {
    std::unordered_map<SongID, Song>      song_db;    // 楽曲DB
    std::unordered_map<AlbumID, Album>    album_db;   // アルバムDB
    std::unordered_map<c7::usec_t, std::pair<ALbumID, SongID>>
                                          history;    // 履歴(なんで辞書)
    std::unordered_map<std::pair<AlbumID, SongID>, int>
                                          price;      // 価格
    std::unordered_map<Localtion, SongID> map;        // えっ、おっ、あっ…
};
```

c7json による定義(まずは手書き)

c7json で定義する場合、次の掲げる表による対応で型を入れ替える(長くなるので std:: は省略)。なお、背景色が黄色の型(c7::json_int, ...)をまとめて c7json基本型、c7::json_object, c7::json_struct から派生した型をまとめて c7json 構造体(型)と呼ぶことにする。また、c7json 型のインスタンスのことを c7jsonオブジェクトと表現する。

通常の型	c7jsonの型 [俗称] (根底の型)	JSON上の表現
整数	c7::json_int [c7json整数型] (int64_t)	整数
タグ付き整数 Tagが異なると代入不可。IDのように同じ整数型でも区別したい場合に最適。	c7::json_tagged_int<Tag> [c7jsonタグ付き整数型] (int64_t)	整数
実数(float, double)	c7::json_real [c7json実数型] (double)	JSON規定の実数
bool	c7::json_bool [c7jsonブール型] (bool)	JSON規定のbool
string, char*	c7::json_str [c7json文字列型] (string)	JSON規定の文字列
c7::usec_t (=int64_t)	c7::json_usec [c7json時刻型] (c7::usec_t)	ISO8601準拠のJSON規定の文字列
バイナリデータ(vector<uint8_t)	c7::json_bin [c7jsonバイナリ型] (vector<uint8_t)	BASE64エンコードしたJSON規定の文字列

要素型Tの配列、vector<T>	c7::json_array<T> [c7json配列型] (vector<T>)	JSON配列(配列要素の型は固定)
pair<Tf,Ts>	c7::json_pair<Tf, Ts> [c7jsonペア型] (pair<Tf, Ts>)	JSON配列(二要素)
tuple<Ts...>	c7::json_tuple<Ts...> [c7jsonタプル型] (tuple<Ts...>)	JSON配列(要素数は固定)
キーの型K,値の型Vの辞書, unordered_map<K,V>	c7::json_dict<K,V> [c7json辞書型] (unorderd_map<K,V>)	JSONオブジェクト キーの文字列は動的。
構造体	c7::json_object を継承したクラス [c7jsonファット構造体型] (根底の型はない) ・メンバー名をキーの文字列に対応づけるため、C++の識別子の規則に従い、かつコンパイル時に固定される。 ・認識できないメンバー名とその値は無関心オブジェクトとして保持することで、書き戻しを可能とする。 ・巨大な構造体の一部のみを定義する場合や、メンバーの追加・変更が発生しうる場合に有効。 ・逆にフットプリントが大きく、巨大配列の要素など大量に存在する型には向かない。	JSONオブジェクト 未定義のキーと値も保存時に復元されるが、逆に typo を検出できない。
	c7::json_struct を継承したクラス [c7jsonスリム構造体型] (根底の型はない) ・メンバー名の規則は同上。 ・認識できないキーとその値は静かに無視する。 ・メンバーの追加が発生しうるが、参照のみの場合に向く。 ・このクラス自体のフットプリントは実質 0 で、巨大配列の要素型としても使える。	JSONオブジェクト 未定義のキーと値は保存時に失なわれる。
	c7::json_strict を継承したクラス [c7jsonストリクト構造体型] (根底の型はない) ・メンバー名の規則は同上。 ・認識できないキーがあるとエラーとする。 ・このクラス自体のフットプリントは実質 0 で、巨大配列の要素型としても使える。 ・大量のオブジェクトが存在し、かつ保存する可能性がある場合に用いる。	JSONオブジェクト 未定義のキーでエラーとなるため、既存のデータが失なわれることは無いが、プログラムの処理が進まなくなる。

この表をもとに、前に定義した構造体を再定義する。c7::json_XXX でメンバーを宣言したあと、c7json_init(....) 内に c7json_member() のそのメンバー名を与えて列挙する。メンバーを宣言しても、c7json_init() に記述しないと認識されない。

```
using SongID = c7::json_int;

struct Song: public c7::json_strict {
    SongID              id;             // 楽曲ID
    c7::json_str        title;          // 楽曲名
    c7::json_int        duration_s;     // 演奏時間
    c7::json_bool       favorite;       // お気に入り
    c7::json_bin        audio;          // 音楽データ

    c7json_init(
```

```
        c7json_member(id),
        c7json_member(title),
        c7json_member(duration_s),
        c7json_member(audio),
        c7json_member(favorite),
    )
};

using AlbumID = c7::json_int;

struct Album: public c7::json_object {
    AlbumID             id;                 // アルバムID
    c7::json_str        title;              // アルバムタイトル
    c7::json_str        artist;             // アーティスト名
    c7::json_usec       release;            // リリース日
    c7::json_array<Song> songs;             // 全楽曲

    c7json_init(
        c7json_member(id),
        c7json_member(title),
        c7json_member(artist),
        c7json_member(release),
        c7json_member(songs)
    )
};

using Location = c7::json_tuple<c7::json_int, c7::json_int, c7::json_int>;

struct Library: public c7::json_object {
    c7::json_dict<SongID, Song>           song_db;    // 全アルバム
    c7::json_dict<AlbumID, Album>         album_db;
    c7::json_dict<c7::json_usec, c7::json_pair<AlbumID, SongID>>
                                          history;
    c7::json_dict<c7::json_pair<AlbumID, SongID>, c7::json_int>
                                          price;
    c7::json_dict<Location, SongID>       map;

    c7json_init(
        c7json_member(song_db),
        c7json_member(album_db),
        c7json_member(hisotry),
        c7json_member(price),
        c7json_member(map),
    )
};
```

この例では、コンストラクタを記述していないが、もちろん定義は自由である。

ファイルとのI/O

ここまでの定義で、Library クラスのオブジェクトを通して、JSON形式でのファイルI/Oなどが可能となる。C++プログラム上で、各メンバーの値をどうやって設定し、あるいは値を得るかは後述するとして、I/Oについては以下のよ

うに関数を使うことで簡単に実現できる。iostream に対応したオーバーロードを使えば、文字列への書き込み、文字列からの読み込みもできるため、プロセス間通信にも適用できる。

```
const char *path = "MyMusicsLibrary.json";
Library lib;

// ファイルからの読み込み
if (auto res = c7::json_load(lib, path); !res) {      // res は c7::result<>
    ... エラー処理 ...
}

... lib のデータを更新 ...

// ファイルへの書き込み
if (auto res = c7::json_dump(lib, path, 2); !res) {   // res は c7::result<>, 2 はインデント数
    ... エラー処理 ...
}
```

ちなみに、この定義で次のような JSON ファイルが入出力される。

```
{
  "song_db":{
    "0":{
      "id":0,
      "title":"Child's Anthem",
      "duration_s":165,
      "favorite":false,
      "audio":""
    },
    ...
  },
  "album_db":{
    "200":{
      "id":200,
      "title":"Toto",
      "artist":"Toto",
      "release":"1978-01-01T00:00:00.000000+09:00",
      "songs":[
        0
      ]
    },
    ...
  },
  "history":{
    "2025-05-20T08:52:06.679272+09:00":[
      7,
      3
    ],
    ...
  },
  "price":{
    "[200,0]":200,
    ...
  },
  "map":{
    "[1,2,3]":9,
    ...
  }
```

```
}
```

c7json 型の値へのアクセス

前表で説明した c7json の型のうち背景が黄色と緑色の型は、呼び出し演算子を適用することで根底の型の参照を得ることができ、これが基本のアクセス方法となる。ここでは、いくつかの型について補足を加える。

*c7json*基本型

c7json基本型については代入演算子、型変換演算子を定義してあるので多くのケースでは呼び出し演算子を省くことができる。c7::json_tagged_int のようなタグ付きの型は、異なるタグ型との代入は禁止される。

```
c7::json_int a, aa;
a = 1;                  // 代入演算子
double b = a;           // 型変換演算子
c7::json_int a2 = a;    // コピーコンストラクタ
aa = a;                 // コピー代入演算子

c7::json_real r = a;    // 根底の型が違っても大丈夫（タグ付きでない場合）
```

*c7json*構造体型

c7json構造体型は C++ の struct で定義されるので、メンバーへのアクセスは struct の場合と変わらない。

```
Song s;
s.title = "Heat of the moment";
s.duration_s = 228;
c7::nseq::mmap_r<uint8_t>("hotm.wav") | c7::nseq::push_back(s.audio());
s.favorite   = true;
```

push_back() 引数の s.audio で呼び出し演算子 () を使用しているのは、この式では根底の型(std::vector<uint8_t>)への型変換演算子が効かないためである。なお、この例はかなり作為的で、次のように書くのが普通かもしれない。

```
s.audio = c7::nseq::mmap_r<uint8_t>("hotm.wav") | c7::nseq::to_vector();
```

ところで、現在の例での Song の定義にはコンストラクタを含めていないため、次のような初期化子を伴った宣言はエラーとなる。

```
// 今の Song の定義ではエラー
Song s {
    SongID{1},
    "Heat of the moment",
    228,
    true,
    c7::nseq::mmap_r<uint8_t>("hotm.wav") | c7::nseq::to_vector(),
    };
```

*c7json*配列型

c7json配列型はコンテナとして最低限の機能(begin, end, size, push_back, emplace_back, operator, clear)しか実装していない。std::vector のフル機能(erase や insert など)にアクセスする場合は、基本通り呼び出し演算子を用いる。

DSN で書いてみる

定義する c7json構造体の数が少なく、メンバーの変更も少ないのであれば手書きのコードでも充分に実用できるが、より多くのJSONオブジェクトやそのメンバーが多いと保守は大変になる。また、手書きでは、コンストラクタを意識して定義しないと前述のように初期化で不便なケースがある。そこで、C++のソースファイルのブロックコメントに簡易な文法でデータ構造を定義し、スクリプトで c7json のソースコードを生成し、埋めこむ方法を説明する。埋め込まれる言語は独自のもので、DSN（Data Structure Notation）と呼ぶ。

これまで用いた例をもとにするが、IDの型は単純なエイリアスでなく、c7jsonタグ付き整数を利用する。.cpp もしくは .hpp に次のようにデータの構造を記述する。この部分は namespace 内に記述しても構わない。

```
... 他の C++コード ...

// タグ型は普通の C++ コードとして定義
struct SongID_tag {};          // 楽曲IDのタグ
struct AlbumID_tag {};         // アルバムIDのタグ

/*[c7json:define]              // C++ブロックコメントで JSONオブジェクトの構造を定義

// ID
SongID  = tagged_int<SongID_tag>;
AlbumID = tagged_int<AlbumID_tag>;

// 楽曲
Song !! {                      // 'Song' と '{' の間の '!!' は「ストリクト構造体」を指定する
    SongID        id;          // 楽曲ID
    str           title;       // 曲名
    int           duration_s;  // 演奏時間
    bool          favorite;    // お気に入り
    bin           audio;       // オーディオデータ(バイト列)
}

// アルバム
Album {
    AlbumID       id;          // アルバムID
    str           title;       // アルバムタイトル
    str           artist;      // アーティスト名
    usec          release;     // リリース日時
    [SongID]      songs;       // 楽曲
}

// ロケーション (x,y,z)
Location = (int, int, int);

// ライブラリ
Library {
    {SongID -> Song}                song_db;     // 楽曲DB
    {AlbumID -> Album}              album_db;    // アルバムDB
    {usec -> <AlbumID, SongID>}     hisotry;     // 履歴
    {<AlbumID, SongID> -> int}      price;       // 価格
    {Location -> SongID}            map;         // マップ
}

*/                             // C++ブロックコメント終わり

//[c7json:begin]
//[c7json:end]

... 他の C++コード ...
```

/*[c7json:define] から始まり、*/ で終わるブロックコメント中に、DSNで JSONオブジェクトの構造を定義する。ここでは雰囲気を感じるだけで良いので、文法説明は省略する。この定義のあとに、//[c7json:begin] と //[c7json:end] の行があるが、この二つは行頭に記述し、これらで狭まれた部分が [c7json:define]ブロックから生成した C++ のコードに置換される。この部分に記述してある内容は完全に置換されるため、ユーザコードをこの中には記述してはならない。

このようにファイルを記述し、「 python3 -m replace_c7json ファイル名 」を実行すると、次のようなコードが生成される。長いので大部分は省略し、c7json 構造体としては Library 部分だけ抜粋してある。

```
    //[c7json:begin]

    using SongID = c7::json_tagged_int<SongID_tag>;

        ~~ 省略 ~~

    using Location = c7::json_tuple<c7::json_int, c7::json_int, c7::json_int>;

    struct Library: public c7::json_object {
        c7::json_dict<SongID, Song> song_db;
        c7::json_dict<AlbumID, Album> album_db;
        c7::json_dict<c7::json_usec, c7::json_pair<AlbumID, SongID>> history;
        c7::json_dict<c7::json_pair<AlbumID, SongID>, c7::json_int> price;
        c7::json_dict<Location, SongID> map;

        using c7::json_object::json_object;

        template <typename T0,
                  typename T1=c7::json_dict<AlbumID, Album>,
                  typename T2=c7::json_dict<c7::json_usec, c7::json_pair<AlbumID,
    SongID>>,
                  typename T3=c7::json_dict<c7::json_pair<AlbumID, SongID>,
    c7::json_int>,
                  typename T4=c7::json_dict<Location, SongID>>
        explicit Library(T0&& a_song_db,
                         T1&& a_album_db=T1(),
                         T2&& a_history=T2(),
                         T3&& a_price=T3(),
                         T4&& a_map=T4()):
            song_db(std::forward<T0>(a_song_db)),
            album_db(std::forward<T1>(a_album_db)),
            history(std::forward<T2>(a_history)),
            price(std::forward<T3>(a_price)),
            map(std::forward<T4>(a_map)) {}

        bool operator==(const Library& o) const {
            return (song_db == o.song_db &&
                    album_db == o.album_db &&
                    history == o.history &&
                    price == o.price &&
                    map == o.map);
        }

        bool operator!=(const Library& o) const { return !(*this == o); }

        c7json_init(
            c7json_member(song_db),
            c7json_member(album_db),
            c7json_member(history),
            c7json_member(price),
            c7json_member(map),
```

```
        )
};

//[c7json:end]
```

見ての通り、これまで手書きしていたコードに加えてコンストラクタや等値・不等値演算子(太字)も追加されている。このコードであれば、初期化子を伴う c7jsonオブジェクトの宣言が可能になる。

部分的な定義での利用

ファイルなどに保存されているJSONデータのうち、その一部のメンバーだけに関心があるアプリケーションというのも想定される。c7json では、必要とする部分だけを定義して用いることができる。例えば、Song うち id, title, duration_s が、Album のうち id, title, artist、release, Library のうち song_db album_db, price が必要な場合は、次のように定義すればよい。

```
// 楽曲
Song {                          // 参照だけなら「Song ! {」と書いて「スリム構造体」を指定するこ
ともできる
    SongID          id;         // 楽曲ID
    str             title;      // 曲名
    int             duration_s; // 演奏時間
}

// アルバム
Album {
    AlbumID         id;         // アルバムID
    str             title;      // アルバムタイトル
    str             artist;     // アーティスト名
    usec            release;    // リリース日時
}

// ライブラリ
Library {
    {SongID -> Song}                    song_db;    // 楽曲DB
    {AlbumID -> Album}                  album_db;   // アルバムDB
    {<AlbumID, SongID> -> int)          price;      // 価格
}
```

この定義で Libraryオブジェクトに c7::json_load でフルスペックのJSONファイルを読み込むことができる。このとき、定義に存在しないメンバーは未関心オブジェクトとして、各 Song, Album, Library オブジェクトに保持される。このあと、price などを変更したあと、c7::json_dump でファイルに書き込んだ場合でも、元のJSONファイルに存在したデータは読み込み前と同じデータとして保存される。

当然ながら、プログラムのオンメモリ上で Libraryデータを新規に構築して、c7::json_dump でファイルに書き込んだ場合は、この定義のメンバーしか出力されない。

API

```
#include <c7json.hpp>
namespace c7
```

c7json構造体定義用のマクロ

```
c7json_init(...)
```

c7json構造体定義の本体部分に一つだけ記述し、c7json のメンバーを基底クラスである c7::json_object / c7::json_struct へ認識させる役割を持つ。このマクロの引数には、後述の c7json_member/c7json_member2 でメンバー情報を列挙する。

```
c7json_member(JSON_NAME)
c7json_member2(JSON_NAME, CXX_NAME)
```

c7json構造体定義内でメンバー情報を生成する。前者は、JSONオブジェクトのメンバーのキー名称とc7json構造体のメンバー名をともに *JSON_NAME* とし、後者は JSONオブジェクトのメンバー名 *JSON_NAME* に対して、c7json構造体上は *CXX_NAME* として定義することを意味する。これは、JSONオブジェクトのメンバー名が、c7json構造体で既定義のメンバー名と衝突する場合に使用する。例えば、load や dump が該当する。

```
c7json_init_declare()
c7json_init_implement(STRUCT_NAME)
```

c7json_init()を宣言と実装に分離する場合に使用する。c7json_init_declare() は c7json_init() がそうであったように c7json構造体の本体内に記述する。c7json_init_implement() は実装側のトップレベルに記述し、*STRUCT_NAME* には c7json_init_declare() を記述した c7json構造体の型名を指定する。

c7jsonの型

```
using json_int  = c7::json::proxy_basic<int64_t>;
using json_real = c7::json::proxy_basic<double>;
using json_bool = c7::json::proxy_basic<bool>;
using json_str  = c7::json::proxy_basic<std::string>;
using json_usec = c7::json::proxy_basic<c7::json::time_us>; // c7::usec_t
using json_bin  = c7::json::proxy_basic<std::vector<uint8_t>>;
```

これらは、それぞれ c7json の基本型の別名である。

```
template <typename Tag>
using json_tagged_int = c7::json::proxy_basic_strict<int64_t, Tag>;

template <typename Tag>
using json_tagged_str = c7::json::proxy_basic_strict<std::string, Tag>;

template <typename Proxy1 typename Proxy2>
using json_pair = c7::json::proxy_pair<Proxy1, Proxy2>;

template <typename... Proxeis>
using json_tuple = c7::json::proxy_tuple<Proxeis...>;

template <typename Proxy>
using json_array = c7::json::proxy_array<Proxy>;
```

JSON配列に対応する c7json配列型の別名である。*Proxy* は c7json の型でなければならない。

```
template <typename KeyProxy, typename ValueProxy>
using json_dict = c7::json::proxy_dict<KeyProxy, ValueProxy>;
```

一般的なJSONオブジェクトに対応する c7json辞書型の別名である。*KeyProxy* は c7json基本型、*ValueProxy* は c7jsonの型でなければならない。

```
using json_object = c7::json::proxy_object;
```

```
using json_struct = c7::json::proxy_struct;
using json_strict = c7::json::proxy_strict;
```

構造体を想定したJSONオブジェクトに対応する c7json 構造体型の基底クラスの別名である。

エンコード／デコード

```
template <typename JsonProxy>
c7::result<> json_load(JsonProxy& proxy, std::istream& in);
template <typename JsonProxy>
c7::result<> json_load(JsonProxy& proxy, const std::string& path);
```

入力ストリーム(*in*)やファイル(*path*)からJSON形式の文字列を読み込み、*proxy* オブジェクトにデコードする。*JsonProxy* は c7json配列か c7json構造体でなければならない。読み出し前の *proxy* オブジェクトの内容は破棄される。

```
template <typename JsonProxy>
c7::result<> json_dump(JsonProxy& proxy, std::ostream& out, int indent = 0);
template <typename JsonProxy>
c7::result<> json_dump(JsonProxy& proxy, const std::string& path, int indent = 0);
```

proxy オブジェクトの内容を JSON形式へエンコードし、出力ストリーム(*out*)やファイル(*path*)に書き込む。*JsonProxy* は c7json配列か c7json構造体でなければならない。ファイルへの書き込みの場合、*path* 文字列に “.old” を付加した名前で書き込み前のファイルが保存される。indentを 1 以上にすると、JSON形式の構造に合わせて適切に改行とインデントが行われる。一方 0 の場合は、改行コードも余分な空白なども出力されないため、プロセス間通信用のデータに適している。

c7json 型共通のAPI

```
c7::result<> load(c7::json::lexer& lxr, c7::json::token& tkn);
```

- 字句解析器 *lxr* と字句オブジェクト *tkn* をもとに、対象c7jsonオブジェクトが対象とするJSON形式をデコードして、c7json内部にデコード結果を保存する。
- 構文解析の常として、*tkn* には既に対象となるJSON形式の最初の字句が読み込まれていなければならない。つまり、c7json配列であれば ‘[‘ が、c7json構造体であれば ‘{‘ が、c7json基本型であればその型の字句が既に読み込まれた状態である。
- 逆にこれら load から戻る時に *tkn* には次の構造の字句は読み込まれていない。c7json配列であれば ‘]’ が、c7json構造体であれば ‘}’ が、c7json基本型は入力のままである。
- 字句解析クラス *lexer* も字句クラス *token* も単独で使用可能であるが、ここでは詳細は述べない。

```
c7::result<> dump(std::ostream out, c7::json::dump_helper& dh);
```

- 出力ストリーム *out* に c7jsonオブジェクトが保持している値を対象のJSON形式にエンコードして出力する。
- 出力ヘルパー *dh* はインデント制御などを行う型。詳細を知る必要はない。

```
void clear();
```

c7jsonオブジェクトが保持している内容をクリアする。クリアした場合、保持している値はデフォルトコンストラクトされた状態となり、後述の check() は false を戻す。

```
bool check() const;
```

c7jsonオブジェクトがデフォルト初期化以外の値を保持しているか確認する。load() で値が設定されたか、コピー/ムーブコンストラクタあるいはコピー/ムーブ代入演算で値が設定された場合は true を戻す。

c7json配列やc7json構造体の場合は、全ての配列要素や全ての構造体メンバーの check() が true を戻した場合だけ true を戻す。これは再帰的に適用される。

```
TYPE& operator()();
const TYPE& operator()() const;
```

[c7json構造体を除き] c7jsonオブジェクトが保持している根底型オブジェクトの参照を戻す。

c7json 配列型 固有のAPI

c7::json_array<*JsonProxy*> 型の固有のメンバー関数について説明する。

```
JsonProxy& operator[](size_t idx);
const JsonProxy& operator[](size_t idx) const;
```

idx 番目の要素への参照を戻す。

```
template <typename U = JsonProxy>
proxy_array& push_back(U&& v);
```

v を配列の最後に追加する。内部の std::vector の push_back に委譲しているだけ。

```
template <typename... Args>
proxy_array& empalce_back(Args&&... args);
```

args で直接構築して配列の末尾に追加する。内部の std::vector の emplace_back に委譲しているだけ。

```
size_t size() const;
```

配列要素数を戻す。

```
JsonProxy& extend();
```

デフォルト構築した要素を配列の末尾に追加し、その要素への参照を戻す。内部の std::vector の emplace_back を引数無しで呼び出し、その返却値をそのまま戻しているだけ。

```
auto begin();
auto begin() const;
auto end();
auto end() const;
```

内部の std::vector の begin と end のイテレータを戻す。

c7json 辞書型 固有のAPI

c7::json_dict<*KeyProxy*, *ValueProxy*> 型の固有のメンバー関数について説明する。

```
auto find(const KeyProxy& k);
auto find(const KeyProxy& k) const;
```

k にマッチする要素を検索し、そのイテレータを戻す。std::unordered_map の find に委譲しているだけ。

```
template <typename U = KeyProxy, typename V = ValueProxy>
proxy_dict& insert_or_assign(U&& k, V&& v)
```

キー *k* が存在しなければ、値を v としての辞書に登録する。存在していれば、値を *v* に置換する。std::unordered_map の insert_or_assign に委譲しているだけ。

```
size_t size() const;
```

配列要素数を戻す。

```
auto begin();
auto begin() const;
auto end();
auto end() const;
```

内部の std::unordered_map の begin と end のイテレータを戻す。

c7json タプル型 固有のAPI

c7::json_tuple<*Proxies...*> 型の固有のメンバー関数について説明する。

```
template <size_t N> constexpr decltype(auto) get();
template <size_t N> constexpr decltype(auto) get() const;
```

std::get<*N*>(*this) を呼び出す。メンバー関数の形式で使えるようにしたもの。

```
constexpr size_t size() const;
```

タプルの要素数を戻す(コンパイル時の定数)。

DSN によるコード生成

DSNの文法

概要でも説明したように、/*[c7json:define] から始まり、*/ で終わるブロックコメント中に、DSNでデータ構造を定義し、その言語から生成したC++コードを挿入する場所を //[c7json:begin] と //[c7json:end] の二つの行末コメントで指定する。この行末コメントは行頭から記述しなければならない。

DSNの文法について説明する。説明で頻出する *IDENT* は C++ の識別子の規則に従うものとする。

- トップレベルは別名定義と構造体定義とが任意の順で並ぶ。
- 別名定義は「 ***IDENT = TYPE ;*** 」で、型名 *TYPE* に対して、*IDENT* という別名を定義する。
- 構造体定義は「 ***IDENT*【!【!】】{ ... }** 」の構造を持つもので、IDENT が構造体の名前となる。IDENT の後に ! がある場合は c7jsonスリム構造体、!! がある場合は c7jsonストリクト構造体、ない場合は c7jsonファット構造帯として定義される。
- 構造体定義の中括弧で囲われた中に、複数の構造体メンバーを定義する。
- 構造体メンバーの定義は「 ***TYPE IDENT ;*** 」で、型 *TYPE* をもつ *IDENT* という名前のメンバーを定義する。
- 型 *TYPE* は簡易言語上の型で、次のように再帰的に定義される。
  - **int, real, str, bool, bin, usec** は型である。これらは、c7json基本型に対応する。
  - 構造体定義の *IDENT*（構造体の名前）は型である。
  - 別名定義の *IDENT*（別名）は型である。
  - *TMPL_TYPE* がテンプレート型で、*CxxType*,... が C++ で定義された型とすると「 ***TMPL_TYPE<CxxType,...>*** 」は型である。現在のバージョンでこの形式が使えるのは *TMPL_TYPE* が **tagged_int, tagged_str** の場合に限る。
  - *TYPE* が型の場合、「 **[*TYPE*]** 」は型である。これは c7json配列となる。
  - *TYPE1*, *TYPE2* が型の場合、「 **< *TYPE1* , *TYPE2* >** 」は型である。これは c7jsonペア型となる。
  - *TYPE1*, *TYPEs*... が型の場合、「 **(*TYPE1* , *TYPEs*...)** 」は型である、これは c7jsonタプル型となる。
  - *TYPE1*, *TYPE2* が型の場合、「 **{ *TYPE1* -> *TYPE2* }** 」は型である。これは c7json辞書型となる。*TYPE1* がキーの型、*TYPE2* が値の型である。
- // による行末コメントが使える。

- マクロ定義や条件コンパイルを含めることはできない。

データ構造定義の分割

データ構造の定義は二つ以上のブロックコメントに分離して書いても良い。この場合、同じ数だけの `//[c7json:begin]` と `//[c7json:end]` の組が必要になる。*N*番目の `/*[c7json:define] ... */` から生成したコードが、*N*番目の `//[c7json:begin] ... //[c7json:end]` に挿入される。

生成するコードの宣言と実装の分離

概要では、単一ファイルに生成する例であったが、`replace_c7json.py` の引数を二つ与えると、宣言と実装を分離することができる。

```
python3 -m replace_c7json HEADER SOURCE
```

最初の引数 `HEADER` のファイルに簡易言語の定義と宣言のコードを挿入する記述があり、二番目の引数 `SOURCE` のファイルに実装コードを挿入する記述がある。どちらも同じ数の挿入個所(`//[c7json:begin] ... //[c7json:end]`)が必要だ。宣言と実装との分離は、定義されるデータ構造が多く、複数のモジュールで `c7json`型を使う場合にヘッダーファイルを軽量にする効果がある。
